University of Alberta

An Experiment in Extracting Programs from Constructive Proofs in a Framework of a Classical Proof Checker

by

Michael A. Michels

©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

Department of Computing Science

Edmonton, Alberta
Spring 1996

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-10741-8

Canada

University of Alberta

Library Release Form

Name of Author: Michael A. Michels

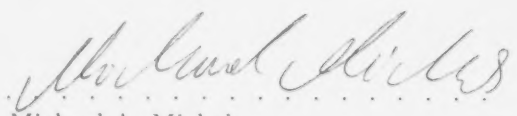
Title of Thesis: An Experiment in Extracting Programs from Constructive Proofs in a Framework of a Classical Proof Checker

Degree: Master of Science

Year this Degree Granted: 1996

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.



Michael A. Michels
25 Flagstone Crescent
St. Albert, Alberta
T8N 1R2

Date: 29/01/1996

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **An Experiment in Extracting Programs from Constructive Proofs in a Framework of a Classical Proof Checker** submitted by Michael A. Michels in partial fulfillment of the requirements for the degree of Master of Science.

Dr. P. Rudnicki

Dr. H. J. Hoover

Dr. B. F. Cockburn

Date: 28/01/1996

Abstract

Programs that are true to their specifications are still an elusive target of software engineering efforts. In the hope of uncovering some insights and gaining new experience we have experimented with using a general, formal proof-checking system for obtaining correct programs. The PC MIZAR system is based on classical logic and an axiomatization of set theory, and includes software for checking proofs written in the MIZAR language.

First, in PC MIZAR, we specify the set of binary sequences, define a number of operations on these sequences, and prove their properties. To each of these operations we provide computational content implemented in Lisp. Then, in PC MIZAR, we specify some constructive inference rules for which we provide computational content—thus obtaining connectives for composing larger programs. Finally, we implement a system which extracts Lisp programs from these parts of MIZAR proofs which have computational content.

We experiment with unary and binary encoding of naturals in binary sequences. Then, we specify and prove properties of arithmetic operations. From the constructive parts of these proofs we extract "almost" correct Lisp programs for arithmetic operations. We then discuss conditions under which the extracted programs are totally correct.

Acknowledgements

I would like to thank my supervisor, Dr. P. Rudnicki, for his help and patience.

I also would like to thank my family, Sonja and Anton, for their support.

Contents

1 Introduction **1**

1.1 Formal methods in program verification 1

1.1.1 External approach to program correctness 1

1.1.2 Internal approach to program correctness 2

1.2 MIZAR-C project 2

1.2.1 Extraction of programs in MIZAR-C 2

1.3 Objectives and motivation for this experiment 3

1.4 The mechanics of the experiment 4

1.5 Overview of the thesis 5

2 PC MIZAR system **6**

2.1 The MIZAR article 6

2.1.1 The text proper section 7

2.2 PC MIZAR system 7

2.3 MIZAR input language 8

2.4 MIZAR abstracts 9

2.5 Main MIZAR Library 9

2.6 The future of the MIZAR system 9

3 Computational extensions to PC MIZAR **11**

3.1 Realizability in MIZ2LISP 11

3.2 The computational model of bit strings 12

3.2.1 Realization of `nil` 13

3.2.2 Realization of `'0` 13

3.2.3 Realization of `'1` 14

3.2.4 Catenation constructor 14

3.2.5 Split destructor 15

3.2.6 Is empty – decision procedure 16

3.2.7 Is zero or one – decision procedure 16

3.3 Realizations of constructive inferences 17

3.3.1 Uniform handling of schemes 17

3.3.2 Implication elimination scheme 18

3.3.3 Or introduction scheme 18

3.3.4 Or elimination scheme 19

3.3.5 Universal statement elimination scheme 20
3.3.6 Existential statement introduction scheme 21
3.3.7 Existential statement elimination scheme 22
3.3.8 Equality scheme 22
3.3.9 Induction scheme 24
3.4 Realization of other PC MIZAR constructs 25
3.4.1 Assumptions 25
3.4.2 Object instantiation 27
3.4.3 Referencing the computational content of objects 28
3.5 Realization with no computational content 28
3.5.1 No content runtime errors 29
3.6 Extending the set of constructive scheme inferences 29

4 Extracting programs from proofs **31**
4.1 Basic `Bits` programs 31
4.1.1 Comparing lengths of two `Bits` 31
4.1.2 Comparing first bits in the string 32
4.1.3 Equality of `Bits` of length 1 33
4.1.4 Equality of `Bits` 33
4.2 `UNat` – unary naturals 33
4.2.1 Unary addition 34
4.2.2 Unary multiplication 35
4.3 Binary naturals 35
4.3.1 Definition of `BNat` mode 35
4.4 Binary addition 37
4.5 Binary subtraction 38
4.6 Binary multiplication 39
4.7 Binary division 40
4.8 Code extraction example – binary multiplication 41
4.8.1 Description of the MIZAR constructs used and their realizations 41
4.8.2 The program and its Lisp realization 43
4.8.3 Invoking the Lisp program 46

5 Implementation notes **48**
5.1 Parsing of the MIZAR language 48
5.2 Processing the built-in base of computational contents 49
5.3 Passing information around – building libraries 50
5.4 Running compiled Lisp code 50
5.4.1 Efficiency considerations 51
5.5 Unfinished Business 51
5.5.1 Access to MIZAR library 51
5.5.2 Renaming references 52
5.5.3 Dealing with case insensitivity of XLISP 52
5.5.4 Fitness of MIZ2LISP software 52

6 Unresolved issues and future directions **53**
6.1 Avoiding runtime errors 53
6.1.1 Brute force 54
6.1.2 Extracting programs without runtime errors 54
6.2 Problem of using programs outside the controlled environment 56

7 Summary and conclusions **57**
7.1 The future of the program extraction from constructive proofs 57

Bibliography **59**

A Definition of bases with computational content **62**
A.1 Preliminaries 62
A.2 Bit strings model 73
A.3 Constructive inference rules 79

B `Bits`, `UNat` and `BNat` **83**
B.1 Bits programs 83
B.2 Unary naturals 84
B.3 Binary naturals 87

C Arithmetic operations on `BNat` **92**
C.1 Binary addition 94
C.2 Binary subraction 95
C.3 Binary multiplication 96
C.4 Binary division 98

Chapter 1

Introduction

Programs which satisfy their specifications are the goal of many software engineering efforts. Yet, it still remains an elusive target, despite the overwhelming effort that goes into software development. This document describes an experiment in extracting correct programs, with respect to their specifications. These programs are extracted from proofs using the PC MIZAR system and some constructive proof techniques. PC MIZAR is a system for checking correctness of classical mathematical proofs.

1.1 Formal methods in program verification

As the need in the area of safety critical applications for error-free software increases, so does increase the need for formal methods in verification of software systems. Only the formal methods can promise us *ultra-reliability* in the computer software.

There are basically two, radically different, approaches to program correctness in current practice. One approach is to study a program as an object in a realm of a formal system. That is, first one writes a program and then one proves its properties. The other one is to use special logics in which one specifies what to compute and the programs are extracted automatically from the proofs of these specifications. In the latter approach the programs are side effects of the reasoning, and in the former the program is the object of reasoning.

1.1.1 External approach to program correctness

The external approach is the one that treats the program as an object. In that approach, one first develops a theory of a programming environment. Next, one proceeds to reason about programs as objects in that theory, just as one would reason about any other objects, like functions or spaces for example, in their respective theories. In that aspect this approach is the classical one since there is nothing special about the terms in mathematics that just happened to be programs.

To use this approach one has only to choose their favourite system for formalizing mathematics and proceed. The significant advantage of this approach is that we know

everything there is to know about the program and its environment.

There is much work involved in developing a programming theory and proving its properties before one can start writing programs. Also the programs, after they have been developed and been proven correct, have to be translated into an implementation language which may or may not be a "true" implementation of the programming environment theory.

The examples of this approach include the specification of the SCM machine in PC MIZAR system ([16], [2]) and implementation of the CLI Stack in Nqthm [12].

1.1.2 Internal approach to program correctness

The internal approach is a newer development that is rooted in the ideas of Martin Löf's type theory [10]. In type theory one defines types using specific constructors that ensure that the derived objects are well defined.

The idea behind this approach is that a program is a side effect of the reasoning in the logic. In this approach the programming environment is defined by the logic of the system. However, that means that we cannot reason about the programs and their environment. In the "internal" system one proves the specifications of what one wants to compute and the extraction of the correct program from the proof becomes the responsibility of the system. This lessens the burden on the programmer but expects more from the system.

Examples of systems that take this approach to program correctness, and that I have had first hand experience with, is COQ project [4], which is based on type theory, and MIZAR-C system ([18], [6]), which uses the theory of bit strings (see section 3.2).

1.2 MIZAR-C project

The MIZAR-C project was originated by J. Hoover, P. Rudnicki and A. Wallenstein. It is based on a first-order natural deduction logic. In its core, it is a framework for experimenting with different logics and theories. The base system can be extended through the addition of axioms and inference rules. Its early incarnation is described in [18]. Since its inception the system has undergone an extensive "face lift". The set of inference rules that are implemented has been revised and expanded. The addition of the "*choice*" rule allows the user to introduce new functional terms into the reasoning. Definitions of predicates have also been added. This latest work has been done by K. Kippen and is described in more detail in [6].

1.2.1 Extraction of programs in MIZAR-C

MIZAR-C is primarily used in experiments which extract programs from proofs written in this system. To extract programs from proofs MIZAR-C uses its *realizability*

mechanism. This mechanism translates constructive inference rules into programming steps.

These ideas were first developed by Kleene [7]. He related mathematical formulas to programs which *realize* them. In other work Curry and Howard restated realizability in terms of natural deduction proofs. One of type theories was developed by Martin Löf [10]. A similar theory forms the base for the Calculus of Constructions [4]. Coquand and Huet used the Calculus of Constructions in the implementation of the COQ system in which programs are extracted from operations on types.

MIZAR-C takes a different approach to program extraction. It does not commit itself to any particular model of computation but allows the user to axiomatize a model of their own. The first experiments dealt with a model which was based on natural numbers in unary representation characterized by the Peano's axioms [18]. The next computational model that was extensively used was based on bit strings (see K. Kippen [6] and E. Ho[5]). Bit strings are finite sequences of binary values. This model of computation was first developed and implemented by myself with help from J. Hoover. The advantage of this model to the ones based on type theory is that the objects of computations can be freely manipulated. That is, there are many ways to construct the same object, and destruction of an object does not have to follow its construction.

The realizability mechanism of MIZAR-C translates only a few of its inferences into program connectives. These are the only ones that actually perform some computation. This approach is labeled *sparse realizability* in MIZAR-C. It tries to determine if a given formula has any computational content, and only if it has, a program for that formula is produced. B. Knight has generalized on this idea in his work [8].

1.3 Objectives and motivation for this experiment

As flexible as the MIZAR-C system is, it still requires a lot of effort from the user in proving the theorems. The bulk of the effort in proving specifications is spent not on the constructive steps, which are simple since they are part of the logic, but on proving the facts about the constructed objects. A new object is constructed from already existing ones by one constructive logical inference; however, proving the properties of the constructed object usually takes many logical steps. How hard it is to prove the properties of an object depends on how highly developed are the notions already formalized in the system. Thus most of the work involves proving ordinary mathematical facts and not constructing objects.

Formal reasoning requires us to think at a very precise level. We can only abstract over the terms that were previously introduced. Thus, "1+1=2" is not "obvious" in a formal system but becomes a theorem to be proven. The existence of all the abstract notions like integers, arithmetic operations, etc., have to be formally derived from the principle theories that define the system. Formalizing highly abstract concepts requires a major effort. It can be compared to pyramid building as we slowly develop one layer of abstraction upon the others all rooted in the founding theories, and

culminating with some profound theorem at the top. From my experience, it looks like the amount of work involved is also comparable.

And here is the big advantage that PC MIZAR holds over other formal systems. It accumulated, over the years, a large number of mathematical facts and abstract concepts that have been proven by many contributing authors. Thus we can spend less time on reinventing mathematics in our proofs. PC MIZAR is a system based on classical logic and set theory which is used for formalizing mathematics. I shall describe PC MIZAR system in a little more detail in chapter 2.

Although the MIZAR-C system is modeled after PC MIZAR, it is only its "poor cousin". In comparison PC MIZAR is a much more mature system, having the benefit of over 20 years of a somewhat turbulent history. The checker is more powerful and there exists an extensive library of previous results. Hence, the motivation for this project: how to "marry" the realizability developed for MIZAR-C with the richness of PC MIZAR?

The objective of this experiment is to extend the PC MIZAR system with the realizability mechanism found in MIZAR-C. As a test suite for these extensions we would like to produce some non trivial programs like the four arithmetic operations: "+", "-", "×" and "÷", on binary encoded natural numbers.

1.4 The mechanics of the experiment

Here is an outline of the logical steps that were involved in this experiment. First the computational model was designed. All computations of the programs are implemented using this model. The model I used for this purpose was based on "*bit strings*". Bit strings are finite sequences of binary values. The model also includes some operations for manipulating these strings: catenation and splitting, and some procedures for examining them: "is-nil" and "0-or-1", see section 3.2.

Once I decided on the bit string model I proceeded with its specification in the MIZAR language. This specification takes the form of several definitions and theorems proven in a MIZAR article. In this way we obtain a set-theoretic description of the set of bit strings and operations on them.

Now, with the basic operations in our model formally defined, we need programming language connectives for writing bigger programs. This is where the constructive inference rules and their realizability come into play. We choose a set of constructive inference rules which will be realized by Lisp expressions implementing our programming connectives. In MIZAR, these rules take the form of schemes of reasoning which are MIZAR constructs that use second order terms (see section 3.3). The realizability mechanism for the constructive inferences have been borrowed from the MIZAR-C system.

At this point, we were ready to write proofs with computational content. First, I wrote some proofs of simple but fundamental theorems about `Bits`. `Bits` is my formalization of bit strings. One of the programs compares the lengths of two strings (with three way output: $<$, $=$, $>$), another one is an equality tester for two bit strings.

Next I proceeded to more challenging "programs". I formalized the unary encoded natural numbers, `UNat`, as a subtype of `Bits`. That they are indeed a true representation of natural numbers is shown by proving the isomorphism between `UNat` and the `Nat` type. `Nat` is MIZAR's formalization of natural numbers. For `UNat` I proved theorems about their addition and multiplication and obtained runnable programs which ran correctly once they were extracted.

Finally, I formalized `BNat`, the binary encoded natural numbers, in terms of `Bits`. I showed the one-to-one correspondence between `BNat` and `Nat` types. Then I defined the four arithmetic operations: addition, subtraction, multiplication and division.

After all these proofs were written with computations "hidden" inside them, the last thing to do was to extract the programs. The extraction is done by the MIZ2LISP tool, which I had to implement from scratch. Given a proof, MIZ2LISP recovers the realizations of the basic operations on `Bits` and combines them into a program according to the usage of the constructive inference rules and their realizations. The programs extracted from MIZAR proofs by MIZ2LISP are coded in Lisp.

An example that illustrates the process of extracting a runnable program from a constructive proof is presented in an almost self-contained section 4.8.

1.5 Overview of the thesis

This thesis is divided into 7 chapters. Chapter 1 provided an introduction. Chapter 2 offers a glance at the PC MIZAR system to give the reader a rough idea what it is about.

The constructive extensions to MIZAR language are discussed in chapter 3. Both, the computational model and the constructive aspects of MIZAR language are presented there. Every item has its computational content explained and examples are given of how to use it. This chapter also introduces the basic program extraction methods used in MIZ2LISP.

Chapter 4 presents MIZAR theorems whose proofs carry programs inside them. It describes the development of unary and binary encodings of natural numbers using bit strings. The major theorems specify the four arithmetic operations on binary naturals: $+$, $-$, $\times$ and $\div$.

Chapter 5 describes the implementation of the software, MIZ2LISP, that translates MIZAR proofs into Lisp code. It also states some of the difficulties I encountered along the way.

In chapter 6, I outline some of the issues left unresolved by this project. The problem of runtime errors is addressed there. It also specifies some possible directions for further development

Finally, the summary and conclusion of this experiment are presented in chapter 7.

Chapter 2

PC MIZAR system

A brief overview of the Mizar project is given by P. Rudnicki in in an electronic document "An Overview of the MIZAR Project"[1]. Here, I present a few condensed excerpts from that document.

The MIZAR project is over 20 years years old. It originated in Poland under the supervision of A. Trybulec. Its purpose is to assist in writing of mathematical papers by checking their correctness. It is rooted in classical logic and set theory which form the bases for all further development.

A. Trybulec designed a language, called MIZAR, in which mathematical articles are written. MIZAR allows for cross referencing to other articles. The author of an article is not permitted to introduce any new axioms to the system. The only two articles which are not checked for validity contain the axiomatizations of Tarski-Grothendieck set theory[15] and strong arithmetics of real numbers[14]. Everything else in the system is proven from these predefined axioms.

The main effort in the recent years of the MIZAR project has been directed at building the library of MIZAR articles. There are almost 400 of them and the number is growing. All components of the system are still evolving, including the language and the checking software. This only underlines that after all these years the MIZAR project is still very much an experimental system.

2.1 The MIZAR article

The source of a MIZAR article is an ASCII file. There are two parts to each article: the environment part and the text proper. The text proper contains statements with justifications of their correctness. MIZAR statements can express new facts or they can define new concepts. The statements of fact, marked as theorems, the definitions of new concepts and schemes of reasoning become parts of the MIZAR library which can be referenced by other articles. In the environment part the author states what parts of MIZAR library are imported for use inside the article. They include symbols,

[1] http://www.cs.ualberta.ca/~piotr/Mizar/MizarOverview.ps

formats of functors, predicates, modes, theorems, definitions and schemes defined in other articles.

2.1.1 The text proper section

There can be several text proper sections in an article each starting with the *begin* keyword. Each text proper section is a sequence of the following items:

1. *Reservation* lets the author specify a default type for identifiers so later on when an identifier is used without explicitly specifying its type the *reserved* one is used.

2. *Definition-Block* defines or redefines MIZAR constructors: term constructors (functors), formula constructors (predicates) and type constructors (modes).

3. *Structure-Definition* introduces new structures. These are entities consisting of a number of fields which are accessed by selectors.

4. *Theorem* marks a proposition for export into the MIZAR library for reference by others.

5. *Scheme*, also an externally visible proposition, allows for occurrences of second order terms.

6. *Auxiliary-Item* is an object that is local to the article.

The goal of writing an article is to prove some useful results that others can use in their work. Only articles verified by the MIZAR processor can be added into MIZAR library.

2.2 PC MIZAR system

PC MIZAR is an implementation of a MIZAR processor. It is implemented on an IBM PC platform under the DOS operating system, hence its name. The power of the MIZAR system comes from its ability to process cross references to other articles in the MIZAR library. A great deal of effort was spent in trying to speed up the process of accessing the MIZAR library. For that purpose all articles that are added to the library are translated into a number of PC MIZAR internal files.

1. *signature files* contain information necessary for parsing occurrences of defined MIZAR phrases.

2. *definition files* store every definition's definiens; the definiendums are stored in signature files.

3. *theorem files* store statements of theorems without their proofs.

4. *scheme files* store statements of schemes without their proofs.

PC MIZAR software is a collection of about 50 programs that manipulate MIZAR articles. The verifier itself is a four step process:

1. *scanner* scans the ASCII file and tokenizes it.
2. *parser* parses the tokenized article.
3. *analyser* checks for proper occurrences of MIZAR expressions.
4. *checker* checks if the premises justify the propositions.

The MIZAR language has a very expressive grammar which allows for overloading of identifiers. The constructors of MIZAR phrases can be defined and redefined many times. All this complexity makes for a very involved parsing and type analysis process.

The checker inference incorporates the following scheme for verifying inferences. When an inference is made of the form

$$\alpha_1, \alpha_2, \ldots, \alpha_k \vdash \beta$$

it is transformed to the following conjunction

$$\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_k \wedge \neg\beta$$

If this conjunction is found contradictory then the inference is accepted. The checker is designed for processing speed and not power. Thus it frequently happens that although an inference is logically correct it still is not accepted. In such circumstances a complex inference should be split into a sequence of smaller ones or even one may have to use a proof structure in order for the checker to accept it.

2.3 MIZAR input language

The MIZAR language is very expressive. From the beginning of the project one of its goals was to develop a rich environment that reflects the traditional ways in which mathematicians work. MIZAR language is so complex that it is way beyond the scope of this thesis to even try to present its basics here. The best way to learn the language is by browsing through the articles of the MIZAR library. There are two user guides by E. Bonarska [3] and more recent one by M. Muzalewski [13]. Also, A. Trybulec describes some of the research aspects of the MIZAR grammar in [17].

In my own experience with PC MIZAR I found that most of the language can be easily grasped by intuition. However, when I needed to use some of the more advanced features, like *attributes*, I encountered some difficulties, especially that there does not seem to be any complete documentation of the language in a human readable form.

2.4 MIZAR abstracts

As part of including an article into the library an abstract of the article is created which contains only the statements of the theorems without proofs. This is done in order to provide users of the library with a reference to its content. These abstracts are typeset using TEX. The whole set of abstracts of the articles contained in the Main MIZAR Library are periodically published in *Formalized Mathematics* [11].

2.5 Main MIZAR Library

The Main MIZAR Library is the most significant advantage of the PC MIZAR system over other formal systems. In 1989 the MIZAR group in Bialystok, Poland, started collecting MIZAR articles and organizing them into a library that is distributed to other MIZAR users. Presently there are almost 400 articles included in this library. The subjects of these articles vary. Most of them are translations of basic mathematics and only a few contain new results.

Right now, every article that is submitted and accepted by the MIZAR processor is included into the MIZAR library. There are programs that try to automate the elimination of repeated or trivial theorems. However, since it is still an experimental system it changes sometimes to the point where some parts of the library must be rewritten to reflect the changes in the system. This caused me some grief and as a consequence I continued using an older version of PC MIZAR system (5.1.03) instead of migrating to the newer one.

There have been a few glitches, however, in the development of the library. Probably the most visible one is that natural numbers in the main library do not widen their type to integers but directly to real numbers.

2.6 The future of the MIZAR system

With respect to the language, the development focuses on type hierarchies and the introduction of an object-oriented mechanism for deriving MIZAR structures. The leader of the MIZAR project, A. Trybulec, says that these changes will have a dramatic impact on the way one does mathematics in MIZAR.

There are also plans for the implementation of an automatic translation of mathematical results between MIZAR and other existing computerized systems for proof-checking mathematics

As the data base of articles is growing so is the problem of maintaining it. Even at its current size, learning the content of the library is a major impediment to using it. Since there is no guide to it, the only way to learn it is through browsing the abstracts. Having access to an "expert" who already knows the library is a definite advantage during the initial stages of learning MIZAR. One reason for duplicated theorems is the fact that sometimes it is easier to prove "again" a simple theorem rather than to find it in the library.

After working with the PC MIZAR system I compiled my own list of "wishes": porting the system out of the PC world, a better user interface, and even having the system stabilized would improve the usability of the system.

Chapter 3

Computational extensions to PC MIZAR

The goal of this experiment is to extract programs from MIZAR proofs. Before we can do that we need to develop a framework of objects and constructive inferences from which to extract the computational contents of our proofs. We are presented with the following three problems:

1. what model of computation to choose,
2. how to express the computational connectives, and
3. how to translate computations hidden inside proofs into runnable programs.

The problem of choosing an implementation language is already solved. The theory tells us that there exists an isomorphism between the lambda calculus and natural deduction [4]. So, I have chosen Lisp for the implementation language since it provides for lambda expressions, and it is what MIZAR-C uses.

3.1 Realizability in MIZ2LISP

The realizability mechanism of MIZ2LISP is borrowed from MIZAR-C system. Actually it is one of the goals of this experiment to adapt MIZAR-C realizability for use with MIZAR. Like MIZAR-C's, MIZ2LISP's realization process does not preserve the Curry-Howard [4] isomorphism between the natural deduction logic and its lambda calculus realization. However, we still have a one-way correspondence from proofs to programs.

The key to MIZAR-C's *sparse realizability* is the notion of a computational content. The formula has computational content if it states something of a computational value; that is, if we can use its results as inputs in further calculations. However, where MIZAR-C goes to lengths to eliminate all non-computational terms from the program, MIZ2LISP only checks for content inside potential lambda expressions. Lambda expressions are created only if some computation has been detected

in its scope. Also the criteria for when a formula receives computational content is much simpler. The formula receives computational content if it is a conclusion of a constructive inference. The inputs to the constructive inferences are not checked for valid computational content. This may lead to runtime errors (see section 3.5.1).

The realizations of formulas in MIZ2LISP are Lisp programs that compute the content of their corresponding formulas. As with MIZAR-C, there are only three types of computational content:

1. *lambda expression* is the content of an assumption (formulas and variables),
2. *object* is the content of a term,
3. *decision* is the content of a disjunction.

The realizability mechanism in MIZ2LISP is the process of extracting these contents and operations on them from proofs in MIZAR articles. The operations on the contents come from the constructive inferences. It is these inferences that form the steps of the extracted programs.

3.2 The computational model of bit strings

Before we can have programs, we need a computational model in which the programs are executed. I chose the model of bit strings for its versatility and easy implementation of binary encodings. The `Bits` type in MIZAR is a formalization of strings of binary values, 0 and 1. There are three components to this model.

- First, there are three basic building blocks from which all other strings are constructed: an empty string and two strings of length one, one containing 0 and one containing 1.
- Second, there are two operations that can be performed on bit strings: catenation and splitting. Catenation is a constructor which is used to build new strings composed from others. Splitting has the opposite effect. It is the destructor which is used to decompose strings in two.
- Third, there are two procedures for inspecting bit strings: one that tells if a given string is empty and one that tells us if a string of length 1 contains 0 or 1.

`Bits` type is the basic data structure on which all other computations are performed. In MIZAR language it is defined in the following way:

```
definition
        mode Bits is Element of {0,1}*;
end;
```

where `{0,1}*` denotes the set of all finite sequences of `0` and `1`. Finite sequences are already a highly developed notion in the MIZAR library.

3.2.1 Realization of nil

The "smallest" term of type Bits is an empty string nil. It is used as such in many places, especially as the base case in induction proofs. Here is its definition in MIZAR.

```
definition :: nil_def
        func nil -> Bits means
:: BITS1: def 3
 it = <φ>;
end;
```

The comments in MIZAR start with "::" and continue to the end of the line. After processing an article PC MIZAR assigns a reference number to each exported item. This reference is printed as part of the abstract of an article so that others can use it when they want to call upon a named fact. The above definition says that nil is a functor of type Bits and it is equivalent to an empty sequence. The computational content of nil is given by this Lisp's expression:

(setq c-nil '())

As it stands, there is no way to reference nil's computational content. Only labels of MIZAR statements can be referenced. This is remedied by the following MIZAR theorem

```
theorem :: BITS1:24 :: C_nil
 nil is Bits;
```

here "BITS1:24" is a reference label of a theorem in article "BITS1.MIZ". I put the other comment to mark correspondence between the numbers assigned by PC MIZAR and the labels I used inside the article. By using the construct "v: v is Bits" I pass the content of the variable "v" to the MIZAR statement labeled "v:". The purpose of this construct is discussed in section 3.4.3. The content of nil is referenced by the label of this theorem.

3.2.2 Realization of '0

The term '0 represents a string of length 1 and 0 as its only element. It is meant to represent a 0 bit. This is its MIZAR definition

```
definition :: Bits_0_def
        func '0 -> non empty Bits means
:: BITS1: def 1
 it = <*0*>;
end;
```

here the quote in '0 is used to differentiate this term from MIZAR's numeral 0. It is given Lisp's

(setq c-zero '(0))

expression as its predefined computational content. This content is linked to a label by the following MIZAR statement

```
theorem :: BITS1:22 :: C_zero
 '0 is Bits;
```

3.2.3 Realization of '1

The term '1 represents a string of length 1 and 1 as its only element. Its MIZAR definition is given by

```
definition :: Bits_1_def
        func '1 -> non empty Bits means
:: BITS1: def 2
 it = <*1*>;
end;
```

and its realization in Lisp is

```
(setq c-one '(1))
```

Again '1's content is linked to a label by the following MIZAR statement

```
theorem :: BITS1:23 :: C_one
 '1 is Bits;
```

3.2.4 Catenation constructor

The catenation is one of the two operations, in the bit strings model, that are used for manipulating the strings. It is the constructor which is used to compose a new string from two others. The catenation operation is defined in MIZAR in the following way:

```
definition :: cat_def
        let p, q be Bits;
        redefine func p ^ q -> Bits;
end;
```

where ^ is the catenation functor. This redefinition is accepted without any further proof because the catenation operation is already defined on finite sequences. Its meaning is that the type of the resulting expression is `Bits` when the "^" functor is applied to arguments of type `Bits`.

Definitions are not processed by the MIZ2LISP extractor. Thus the predefined computational content of this operation has to be referenced in some other way. The following theorem is given the built-in realization of catenation.

```
theorem :: BITS1:28 :: C_cat
 for p, q being Bits ex r being Bits st r = p ^ q;
```

And this is the Lisp expression that realizes the catenation operation.

```
(setq c-cat
      (lambda (x)
        (lambda (y)
          (list (append (content x) (content y)) t))))
```

The intent is that the second `Bits` term is appended to the end of the first one. We return a list because of the realization of an existential statement, see section 3.3.6. The "*content*" function checks for empty realizations, "*t*", and crashes when one is found.

3.2.5 Split destructor

The destructor, split operator, of the bit string model is the counterpart to the catenation operator. It is used to split `Bits` terms into their substrings. In the case of split operation there were no equivalent notions available and ready for reuse in the MIZAR library. So, the following definition of the split functor was my first significant effort in this project involving MIZAR. Note that this functor is total on its arguments.

```
definition :: split_def
        let p, q be Bits;
        func p ^' q -> Element of [: {0,1}*, {0,1}* :] means
:: BITS1: def 4
 it'1 ^ it'2 = p &
        (len q ≤ len p implies len it'1=len q & len it'2=len p-len q) &
                (len q ≥ len p implies len it'1 = len p & it'2 = nil);
end;
```

This definition uses a `Bits` term to specify where to split the string. This is because the only objects in our computational model are bit strings and it would be impossible to use anything else. The result of splitting `p` is an ordered pair of `Bits` whose first element is the first `len q` items of `p` and the second one is the rest of `p`. In the above definition, "`^'`" denotes the split functor. The first argument is the string to be split and the second argument specifies the split point. In the MIZAR definition, `it` stands for the result of the functor applied to its arguments, `it'1` is the first element of the pair forming the result and `it'2` is the second one. `len` is the "length of" functor already defined for finite sequences.

As with catenation, the predefined realization of split operation is given to a separate theorem.

```
theorem :: BITS1:29 :: C_split
 for p, q being Bits ex r1, r2 being Bits
        st p ^' q = [r1, r2];
```

Since there is no built-in implementation of Cartesian product in our string model, the result of split has to be taken apart. The predefined realization of the split destructor is this Lisp expression:

```
(setq c-split
    (lambda (x)
      (lambda (y)
        (if (content x)
            (if (content y)
                (let ((ret (funcall (funcall c-split (cdr x)) (cdr y))))
                  (list (cons (car x) (car ret)) (cadr ret)))
              (list nil (list x t)))
          (list nil (list nil t))))))
```

As you can see, the realization is as equally awkward as the definition. The peculiarity of this code is rooted in the nested existential statements. Each existential formula is realized by a pair. Here, the first element is the object realizing the term bound by the first quantifier and the second element is the list containing the realization of the second quantifier and the remainder of the formula. Existentially quantified statements are discussed in section 3.3.6.

3.2.6 Is empty – decision procedure

There are two built-in procedures in the bit string model that allow us to examine the string. The first one is a procedure that tells us if a given string is empty or not. This information is captured by the following theorem.

```
theorem :: BITS1:26 :: C_nil_or_not
 for b being Bits holds b = nil or b <> nil;
```

Note that this theorem is obvious to the PC MIZAR system and is accepted by the PC MIZAR processor on its own. The above theorem is realized by the following Lisp expression:

```
(setq c-nil-decide
    (lambda (x)
      (if x (list nil t) (list t nil))))
```

the first element of the returned pair corresponds to "`b = nil`" and the second to "`b <> nil`" of the above MIZAR theorem.

3.2.7 Is zero or one – decision procedure

The second of the two inspection procedures allows us to examine the individual "bits" in the bit strings. This MIZAR theorem corresponds to a program that decides if a given `Bits` term of length 1 is `'0` or `'1`.

```
theorem :: BITS1:27 :: C_unit_0_or_1
 for b being Bits st len b = 1 holds
        b = '0 or b = '1;
```

Its predefined realization in Lisp is the following lambda expression:

```
(setq c-01-decide
    (lambda (x)
      (lambda (y)
        (if (equal c-zero (content x)) '(t nil) '(nil t)))))
```

where the first element of the returned pair corresponds to "`b = '0`" and the second one to "`b = '1`" of the MIZAR theorem.

3.3 Realizations of constructive inferences

The constructive inferences correspond to the programming connectives which operate on the contents of their premises and produce the content of their conclusions. As such, the predefined realizations of the constructive scheme inferences are the elementary programming steps in which all other programs are expressed.

Schemes in the MIZAR language allow for manipulating second order terms. This provides means for abstraction over types, functions and predicates in MIZAR proofs. This feature allowed me to employ schemes of MIZAR as the encoding mechanism for the constructive inferences. The constructive schemes "show" the evidence for their conclusions. This evidence is constructed from the premises given to the scheme. It is this construction process that becomes the realization of a constructive scheme.

There are several basic constructive schemes that have their realizations predefined by MIZ2LISP. They correspond to basic constructive inferences in the natural deduction logics. They are described in the remainder of this section.

3.3.1 Uniform handling of schemes

The constructive inferences are made through the use of schemes that have realizations. The propositions in MIZAR which are justified by the use of constructive schemes are realized by the application of programming connectives (combinators) behind the schemes to the realizations of their premises.

The program behind a scheme is applied to the realization of the first premise. Then the result is applied to the realization of the second premise, if any exists, and this result is applied to the realization of the third premise, if any, and so on. This continues for all the premises. For example, the following scheme inference

α `from scheme_ref( label1, label2 );`

is realized by the following Lisp code

(funcall (funcall `scheme_ref label1:` *)* `label2:` *)*

and becomes the computational content of the formula α. Here, *scheme_ref* stands for realization of `scheme_ref` scheme, and *label1:* and *label2:* are realizations of the MIZAR statements labeled `label1` and `label2`.

3.3.2 Implication elimination scheme

Implication corresponds to execution of two blocks of code consecutively. That is, first compute the antecedent and then use the results to compute the consequent. In this sense implications subdivide a program into subroutines. Here is the definition of the *Implication Elimination* scheme

```
scheme C_I_E { P[], Q[] }:
       Q[]
provided
      P[] implies Q[] and
      P[];
```

This says that `P[]` and `Q[]` are any propositions. The premises to this scheme are listed after the `provided` keyword and are separated by the `and` keyword. The computational content of this scheme is to apply the program behind "`P[] implies Q[]`" to realization of `P[]`.

```
(setq c-implication-elim
     (lambda (x)
      (lambda (y)
       (funcall (content x) y))))
```

Example:

```
a  from C_I_E( imp, ant )
```

where `imp` is the label of the implication and `ant` is the label of the antecedent. This inference produces the following Lisp code.

```
(funcall (funcall c-implication-elim imp:) ant:)
```

where *imp:* and *ant:* are realizations of the corresponding MIZAR formulas.

3.3.3 Or introduction scheme

Disjunction expresses a decision making mechanism which in turn is used by the *Case Analysis*. There are two ways to introduce a disjunction constructively. One is by showing its left disjunct and the other one is by showing its right disjunct. Hence we have two schemes, one for stating disjunction's left hand side:

```
scheme C_O_I_1 { P[], Q[] }:
       P[] or Q[]
provided
      P[];
```

and one for stating disjunction's right hand side:

```
scheme C_O_I_r { P[], Q[] }:
        P[] or Q[]
provided
      Q[];
```

Their corresponding realizations are

```
(setq c-or-intro-left
      (lambda (x)
        (list x nil)))
```

and

```
(setq c-or-intro-right
      (lambda (x)
        (list nil x)))
```

The realizations of disjuncts are passed along to the realization of the disjunction because they may be used by their respective cases when the disjunction is used in *Case Analysis*. More on that in section 3.3.4.

Examples:

```
α  from C_O_I_l( left );
α  from C_O_I_r( right );
```

The above justifications produce the following realizations of their conclusions:

```
(funcall c-or-intro-left left:)
(funcall c-or-intro-right right:)
```

where *left:* and *right:* correspond to the realizations of the corresponding premises.

3.3.4 Or elimination scheme

Case Analysis reasoning is captured by the following scheme.

```
scheme C_O_E { P[], Q[], R[] }:
        R[]
provided
      P[] or Q[] and
      P[] implies R[] and
      Q[] implies R[];
```

The semantics of the Case Analysis inference is that we have two ways of constructively inferring the same conclusion. We know that if the preconditions of one of them are true, then we can constructively arrive at the conclusion by following the respective path. But it is not necessary to know which path is taken. This represents the element of non-determinism in the reasoning since we do not know which path

will be taken a priori. In programming terms, this reasoning represents a decision point in a computation. Depending on the outcome of the decision, which comes from the realization of the disjunction, one of the alternative paths in the program is taken.

The following Lisp code realizes this scheme.

```
(setq c-or-elim
    (lambda (pair)
      (lambda (x)
        (lambda (y)
          (if (car (content pair))
              (funcall x (car pair))
            (funcall y (cadr pair)))))))
```

The *pair* is the realization of the disjunction from section 3.3.3. Based on which disjunct is true, implication elimination is performed using the respective case and disjunct.

Examples:

```
a  from C_O_E( disj, left_case, right_case );
```

The above inference produces this Lisp realization of its conclusion

(funcall (funcall (funcall c-or-elim disj:) left_case:) right_case:)

where *disj:*, *left_case:*, and *right_case:* are realizations of the correspondingly labeled statements in MIZAR text.

3.3.5 Universal statement elimination scheme

Universally quantified formulas represent functions. Thus, discharging a universal statement is equivalent to a function call in a program. Below is the definition of the universal statement elimination scheme.

```
scheme C_U_E { D() -> non empty set,
               B() -> Element of D(),
               P[Element of D()] }:
        P[B()]
provided
      for b being Element of D() holds P[b] and
      B() is Bits;
```

here `Element of D()` is a type term parameterized by `D()`. I use the proposition "`B() is Bits`" to refer to the computational content of term `B()` since only references to labeled statements are allowed when using a scheme justification (see section 3.4.3). The realization of the above scheme is the following Lisp expression:

```
(setq c-universal-elim
    (lambda (x)
     (lambda (y)
      (funcall (content x) (content y)))))
```

Note that this realization looks the same as the realization of implication elimination scheme. Since Lisp is an untyped language, the difference in types between the two schemes that is evident in MIZAR language is lost here.

Examples:

```
a  from C_U_E( uni_s, term );
```

produces the following realization

(funcall (funcall c-universal-elim uni_s:) term:)

where *uni_s:* and *term:* are the realizations of a universal statement and a term "is Bits" statement respectively.

3.3.6 Existential statement introduction scheme

Locally instantiated variables cannot be a part of the conclusion of a reasoning in MIZAR. Otherwise, the conclusion, which is used outside of the local reasoning's scope, would refer to uninstantiated variable. An existential quantifier is used to bind the variables occuring within a formula. Then the formula can become a conclusion. Thusly exported formula does not include any occurrences of uninstantiated variables. In realizability terms, this couples the computational content of the variable being quantified with the content of the formula. For that reason, existential quantifier introduction is compared to the return statement in a program. Here is the definition:

```
scheme C_E_I { S() -> non empty set,
        A() -> Element of S(),
        P[Element of S()] }:
        ex b being Element of S() st P[b]
provided
      P[A()] and
      A() is Bits;
```

and again, "`A() is Bits`" is a "place holder" for the computational content of the `A()` term. The realization of this scheme is given in the following Lisp code.

```
(setq c-existential-intro
    (lambda (x
     (lambda (y)
      (list (content y) x))))
```

The realization of the existential statement contains first the realization of the quantified variable and then the realization of the formula.

Examples:

```
ex v st ...  from C_E_I( form, var );
```

Here `form` refers to the formula containing local variable and `var` is the label of that variable's content binding "... `is Bits`" statement. This justification results in the following realization of the conclusion

(funcall (funcall c-existential-intro form:) var:)

where *form:* and *var:* are realizations of the respective MIZAR statements.

3.3.7 Existential statement elimination scheme

MIZAR handles existential quantifier elimination through a "`consider`" statement. See section 3.4.2 for discussion of that construct. This statement is justified by an existentially quantified formula. The existential statement elimination scheme arises from the need to reference an existential statement and pass its computational content at the same time. Thus this scheme is an identity that MIZ2LISP can recognize and process appropriately. Here is its definition:

```
scheme C_E_E { S() -> non empty set,
                P[Element of S()] }:
       ex a being Element of S() st P[a]
provided
      ex a being Element of S() st P[a];
```

And here is its realization in Lisp

(setq c-existential-elim
(lambda (x) (content x)))

Examples:

```
consider v such that ...  from C_E_E( exists );
```

where `exists` is the label of an existentially quantified statement. This scheme produces the following realization in Lisp:

(funcall c-existential-elim exists:)

where *exists:* is the realization to be passed along.

3.3.8 Equality scheme

Sometimes in MIZAR it is necessary to manually change the type of a variable. This is done in the following way

```
reconsider v' = v as Another_Type;
```

In the processing of this statement, MIZ2LISP passes the computational content of `v` to `v'` since they both reference the same object in MIZAR. Then we continue to show some results "`P[v']`" using `v'`, a "shadow" of `v`. However since `v'` is a local variable it cannot occur in the conclusion. And anyway, we are probably interested in "`P[v]`". Thus we have to express the results using `v`, "`P[v]`". MIZAR knows that these two terms are the same and has no problems accepting a straight forward justification. However, for the same reasons as in section 3.3.7, we need a constructive scheme justification to let MIZ2LISP know about the passing of computational content. Unfortunately, none of the other scheme inferences fit here, so we need a new scheme to express the equality among the two, seemingly different, variables and the propositions that use them. We cannot just use an identity since to MIZAR the difference in types makes the propositions look different.

Here is the equality scheme that solves the above dilemma. The extensive type parameterization in this scheme allows for widening of types. I am not sure at this time if the version accommodating narrowing of types is also needed so I left it out.

```
scheme C_EQ { D() -> non empty set,
              A() -> Element of D(),
              D'() -> non empty Subset of D(),
              A'() -> Element of D'(),
              P[Element of D()] }:
       P[A'()]
provided
      A'() = A() and
      P[A()];
```

The realization of this scheme in Lisp is an identity since there are no types in Lisp. But to reflect the "guarding" "`A'() = A()`" premise in the scheme the realization takes two arguments.

```
(setq c-equality
    (lambda (x)
      (lambda (y) (content y))))
```

Examples:

```
a  from C_EQ( eq, prop );
```

The above inference will produce the following Lisp realization

(funcall (funcall c-equality eq:) prop:)

where *eq:* and *prop:* realize corresponding MIZAR statements.

3.3.9 Induction scheme

Induction is the "work horse" behind all the extracted programs from MIZAR proofs. It is the only means for computing something more substantial than just a few static expressions. MIZ2LISP extensions to MIZAR use the following version of complete induction.

```
scheme C_Comp_Ind { D() -> non empty set,
                    F(Element of D()) -> Nat,
                    P[Element of D()] }:
       for a being Element of D() holds P[a]
provided
 for a being Element of D() st for a' being Element of D() st F(a') < F(a)
       holds P[a'] holds P[a];
```

where `Nat` is the MIZAR type which represents natural numbers. This scheme can be applied to any type: "`Element of D()`", using any partial ordering on it: "`F(Element of D()) -> Nat`". The only premise to this scheme is the inductive step. Note that the inductive step incorporates the base cases in it.

The realization of this scheme employs recursive calls to the inductive step with "smaller" values through the references to the inductive hypothesis. The recursion is guaranteed to terminate with base cases since the inductive hypothesis cannot be used in their proofs. The base cases inside the inductive step have to be derived using other means. Here is the Lisp code for the induction scheme.

```
(defun c-ind-prop (istep)
   (lambda (x) ;; recursive function from induction
     (funcall ;; call inductive step with . . .
      (funcall (content istep) (content x)) ;; the argument and . . .
      (lambda (y) ;; inductive hypothesis
        (setq c-ind-count (+ c-ind-count 1)) ;; count recursive calls
        (lambda (y-lt-x) ;; check for ( y < x )
          (funcall (c-ind-prop istep) (content y) ))))))) ;; recursive call

(setq c-induction (function c-ind-prop)) ;; for use with `funcall`
```

Examples:

```
a  from C_Comp_Ind ( IS );
```

This application of Induction is straightforward. It is the inductive step that contains all of the computations. `IS` is the label of the inductive step. It produces the following Lisp realization

```
(funcall c-induction IS:)
```

where *IS:* is the realization of the inductive step.

3.4 Realization of other PC MIZAR constructs

The constructive schemes of the previous section correspond to the "execution" steps in the extracted programs. That is, they specify the logic of the program. However, they do not provide the means for dealing with issues of storage allocation. Whenever we declare an object in a program, we reserve the space for it.

There are three types of objects in MIZAR that require storage for their counterparts in the extracted programs: functions, formal parameters and local variables. Opening a new scope of reasoning in MIZAR text corresponds to a function declaration for that scope. Assumptions inside that scope correspond to declarations of formal parameters for that function. And finally, the instantiation of "computed" MIZAR terms corresponds to local variable declarations. These constructs and their realizations are the subject of the following discussion.

3.4.1 Assumptions

Assumptions in MIZAR correspond to function abstractions. There are two types of them: assumptions that introduce variables, which are abstractions over variables, and assumptions of formulas which abstract over formulas. The former introduces a universal statement in MIZAR, the latter introduces an implication. Assumptions in MIZAR can only occur inside some open scope.

3.4.1.1 Scopes of reasoning in MIZAR

MIZAR allows for multilevel scopes of reasoning. In most cases new scopes of reasoning open with "`now`" or "`proof`" keywords and terminate with "`end`". There are some other ways to open and close a new scope of reasoning but they are ignored by the MIZ2LISP translator and thus do not concern us here. The "`proof`" keyword occurs exclusively after a proposition to indicate that a more involved justification follows. The conclusion of "proof" scope is thus fixed by the preceding proposition. A new scope that opens with "`now`" is a new compound statement. We are free to conclude anything we can inside it. The compound statement is interpreted as a universal statement, an implication or the combination thereof, depending on the reasoning inside its scope.

The conclusions of the reasoning are indicated by "`thus`". It is possible to state several conclusions in a MIZAR reasoning. MIZAR groups them into one conjunction. However, conjunctions do not have any computational content and MIZ2LISP just ignores them. As a result only the last conclusion is taken into account by the MIZ2LISP processor. Our purpose here was to simplify things for ourselves and the simplification introduced here cannot be exploited later since there is no constructive scheme that operates on conjunctions.

3.4.1.2 Universal statement introduction

Introduction of free variables in a MIZAR reasoning is done by the use of the `let` statement. MIZAR binds these free variables to the conclusion of the reasoning with universal quantifiers. The following fragment of MIZAR reasoning

```
for v, w holds P[v,w]
proof
    let v;
    let w;
    ⋮
    thus P[v,w] from ...; ::  constructive inference here
end;
```

results in the following realization

(lambda (v) (lambda (w) ... #R(`P[v,w]`*)))*

where "*#R(*`P[v,w]`*)*" stands for realization of the reasoning's conclusion. In a realization of a given reasoning, a new lambda expression is introduced for each introduction of a free variable.

3.4.1.3 Implication introduction

Assumption of a proposition in the scope of a given reasoning turns its conclusion into an implication. The assumed proposition becomes the antecedent and the original conclusion forms the consequent. Formula and free variable assumptions can be intermixed with each other. Here is an example of an implication introduction.

```
for v st P[v] holds for w st Q[v,w] holds R[v,w]
proof
    let v;
    A1: assume P[v];
    let w;
    A2: assume Q[v,w];
    ⋮
    thus R[v,w] from ...; ::  constructive inference here
end;
```

where `A1:` and `A2:` are labels of the assumptions. The above text fragment has the following computational content

(lambda (v)
(lambda (`A1:`*)*
(lambda (w)
(lambda (`A2:`*) ... #R(* `R[v,w]`*)))))*

where *#R(*`R[v,w]`*)* stands for the realization of the conclusion. Since there are no types in Lisp, there is no distinction between the two types of assumptions in MIZAR language.

Unlike MIZAR-C, MIZ2LISP realizes all implications with lambda expressions as long as their conclusions have computational content. MIZAR-C tries to optimize on the number of formal parameters a function has by not including parameters which are not used constructively. MIZ2LISP does not share this "feature" with MIZAR-C. The main reason is that by keeping all formal parameters, the realizations reflect their proof counterparts. Then, the extracted Lisp code closely resembles the uses of the theorems with computational content in the proof since there is no "missing" implication eliminations corresponding to the excluded "non-constructive" parameters.

I strived for clarity and not runtime performance. I found that the kind of optimization MIZAR-C does by reducing the number of formal parameters to decrease the number of functions calls is ill served at this level of development. As long as other issues of correctness are still unresolved, I think that the increase in the clarity of extracted Lisp programs overrides any marginal gains in runtime performance.

3.4.2 Object instantiation

Object instantiation compares with an assignment statement of programming languages. MIZAR terms that result from the previous computations are "assigned" to the local variables via "`consider`" construct. MIZAR's "`consider`" statement introduces a local variable into the reasoning. This variable is an instantiation of some MIZAR term whose instance is attached to it. These instantiations are justified by the equivalent existential formulas. Here is an example of the `consider` statement.

```
now
    :
    consider v such that
    Pv: P[v] from α; ::  constructive inference
    :
    thus β from ...; ::  constructive inference
end;
```

where `v` is the instantiated variable, "`Pv: P[v]`" is the "stripped" remainder of the justification, and β is the conclusion of the reasoning. The instantiation of an existentially bound variable decouples it from the rest of the formula. This is exactly the computational content behind this construct. Here is the Lisp translation of the above MIZAR reasoning.

```
( ...
  (let ( (v (car #R(α))) )
      (let ( (Pv: (cadr #R(α))) )
          ( ... #R(β) ) ) ) ... )
```

where *#R(α)* is the realization of the justification that gives rise to the existentially quantified formula and *#R(β)* is the realization of the conclusion in the above reasoning. The last ... marks closing parenthesis for the scopes opened before the occurrence of the consider statements. The above Lisp code decomposes the computational content of the existential formula into realization of the variable *v* and the realization of formula "*Pv:*".

3.4.2.1 `reconsider` statement

The `reconsider` statement, in a similar way to the consider statement, introduces a local instance of a variable into the reasoning. Only this time it is just a shadow of an existing term. This construct tells the MIZAR processor to look at the given term as if it were of another type.

```
reconsider v' = v as Another_Type;
```

This only introduces an alias for an existing term and thus MIZ2LISP handles this case internally.

3.4.3 Referencing the computational content of objects

Since only the labels of MIZAR statements can be used as arguments to schemes, we need a way of assigning the content of MIZAR terms to MIZAR statements. MIZ2LISP implements this by assigning the content of a variable to the following "special" statement in which the variable occurs.

```
v: v is Bits;
```

The type `Bits` is of no consequence here. But, since it is the type of the underlying data structure on which all computations are performed, all MIZAR objects that have any computational content had better widen to this type. The `Bits` type then serves as an extra check that the constructive inferences are using terms of proper type. This is the realization of the above statement.

(... (let ((v: #R(v))) (...)) ...)

Here, ... mark the realization of the external context in which the above MIZAR statement occurs. *#R(v)* stands for the realization of variable `v`.

3.5 Realization with no computational content

Every other formula and term in a MIZAR article that was not mentioned above has no computational content. All MIZAR items that do not have any computational content are implicitly realized by Lisp's "*t*" to state that the facts they express are true. I call it an empty realization. The analogy from programming would be a constant expression that is not being referenced by anything and thus is not contributing in any way to the computation.

3.5.1 No content runtime errors

Empty realizations may cause execution errors when used in places where computational content is expected. In its present form MIZ2LISP does not check for this possibility and will produce programs that will crash if it occurs. Here is a "real life" example from my experiment.

```
IS: for u being UNat st
        (for u' being UNat st len u' < len u holds
            B2N u' = 0 & for a being Bits holds
                B2N(u' ^ a) = B2N a)
        holds B2N u = 0 & for a being Bits holds
                B2N(u ^ a) = B2N a
proof
 ⋮ :: non-constructive proof
end;

theorem :: UNat_B2N
UNat_B2N: for u being UNat holds
B2N u = 0 & for a being Bits holds B2N(u ^ a) = B2N a
from C_Comp_Ind(IS);
```

But "IS" statement, that is referenced by the induction scheme, has no computational content and thus this induction inference was translated by MIZ2LISP into the following Lisp code.

```
(setq UNat_B2N:
  (funcall c-induction t))
```

Now, if we go back to section 3.3.9 we will see that built-in function "*c-induction*" expects a lambda expression as its argument. This code will "crash" when Lisp will try to perform *funcall* on *t*. This theorem was never meant to have any computational content and is not used by other programs in a constructive fashion. The use of constructive scheme here was just convenient.

3.6 Extending the set of constructive scheme inferences

Since the handling of schemes that have computational content is independent of the programs that realize them, one can easily define more schemes that have "non-empty" realizations. Here is an example:

```
scheme C_UI_E { D() -> non empty set,
                A() -> Element of D(),
                P[Element of D()], Q[Element of D()] }:
        Q[A()]
```

```
provided
A:      for b being Element of D() holds P[b] implies Q[b] and
B:      A() is Bits and
C:      P[A()]
proof
K:      P[A()] implies Q[A()] from C_U_E(A,B);
        thus
        Q[A()] from C_I_E(K,C);
end; ::
```

The above scheme definition combines the eliminations of a universal statement and of an implication into one scheme of inference. The program that invokes the realizations of these inferences as its computational content is given below.

```
(setq C_UI_E
  (lambda ( A: )
    (lambda ( B: )
      (lambda ( C: )
        (let ((K: (funcall (funcall c-universal-elim A:) B:)))
          (let ((thus: (funcall (funcall c-implication-elim K:) C:)))
            thus: ))))))
```

Chapter 4

Extracting programs from proofs

The goal of this experiment is to mechanically translate programs embedded in the MIZAR articles into *Lisp*. The programs chosen for this task are the four arithmetic operations: addition, subtraction, multiplication and division, performed on natural numbers in binary encoding. These programs represent a nontrivial bit of computation that is wholly derived from the axiomatization of the set theory and the theory of real numbers. That is about as formal as one can get in classical logic. Also, implementation of these operations shows that we can manipulate arbitrarily large binary encoded natural numbers which, for example, could be used to simulate all finite programs on a Turing machine. If we can do that then we can compute anything that is computable using this bit strings model, at least in theory, and we can do it efficiently.

This experiment concluded in 45k words in 9k lines in 113 theorems in 3 MIZAR articles of formal proofs (in painful detail) and in 15.5k words in 4.5k lines of C++ and related source code. The PC MIZAR articles translated into 12k lines and 39.4k words of Lisp.

4.1 Basic `Bits` programs

The following are theorems whose realizations were the first programs extracted using `Bits`.

4.1.1 Comparing lengths of two `Bits`

The program that compares the lengths of two `Bits` has been extracted from the proof of the following theorem.

```
theorem :: BITS2:1 :: C_len_decide
 for p being Bits holds for q being Bits holds
        len p < len q or len p = len q or len p > len q;
```

The program behind this theorem uses induction over the first argument. While both strings are not empty they are decreased in length and passed to the recursive call.

The realization of this theorem groups the disjuncts to the right. This forms a list of cases that can be examined one by one from the left. The realization of this theorem has the following form. By "has the form" I mean that the following describes how many arguments the program takes and what kind of expression it returns. This applies to the remainder of this chapter.

```
(lambda (p)  (lambda (q)
   (#R( len p < len q)
    (#R( len p = len q) #R( len p > len q) ) ) ) )
```

where *#R(...)* denotes the realization of the given MIZAR item. One has to be careful when using this theorem: other computations must not decompose its result by grouping disjunctions to the left or the extracted program will crash. This disturbing occurrence is a result of the PC MIZAR system being too "smart" when it comes to processing schemes. From the beginning of this project it was assumed that all type checking would be done by the MIZAR processor. It was only in later stages of the experiment that it was discovered that this was not so.

The proof of the following theorem uses the results from the previous one.

```
theorem :: BITS2:2 :: C_len_decide2
 for p, q being Bits
        holds len p < len q or len p ≥ len q;
```

The program behind this theorem just calls the program of the previous theorem and reports the result back.

The program has the following form.

```
(lambda (p)
  (lambda (q)
    (#R( len p < len) #R( len p ≥ len q) ) ) )
```

4.1.2 Comparing first bits in the string

The following theorem's proof checks if the given pair of `Bits` agree in the first position.

```
theorem :: BITS2:3 :: C_eqf_or_not
 for a, b being Bits st a <> nil & b <> nil
        holds a.1 = b.1 or a.1 <> b.1;
```

The program of this theorem extracts the first "bits" of both strings and then compares them. It has the following form

```
(lambda (a)
  (lambda (b)
    (lambda ( a <> nil & b <> nil)
      (#R( a.1 = b.1) #R( a.1 <> b.1) ) ) ) )
```

4.1.3 Equality of Bits of length 1

The proof of this theorem produces the program that compares Bits of length one.

```
theorem :: BITS2:4 :: C_unit_eq_or_not
 for a, b being Bits st len a = 1 & len b = 1
        holds a = b or a <> b;
```

The program is just a two-way, two-level deep case analysis. It has the following imprint.

```
(lambda (a)
  (lambda (b)
    (lambda (len a = 1 & len b = 1)
      (#R( a = b) #R( a <> b) ) ) ) )
```

4.1.4 Equality of Bits

The following theorem is realized by a program which checks if the given Bits are the same or not.

```
theorem :: BITS2:5 :: C_eq_or_not
 for p, q being Bits holds p = q or p <> q;
```

The program strips off the first "bits" of the arguments and, if they are the same, it passes the remainders of the arguments to the recursive call. This is its Lisp form.

```
(lambda (p)
  (lambda (q)
    (#R( p = q) #R( p <> q) ) ) )
```

4.2 UNat – unary naturals

The UNat type in MIZAR is a unary representation of natural numbers. Here is its definition. First we define the set and then the mode. A mode is MIZAR's counterpart of a type in a programming language.

```
definition :: UNAT_def
        func UNAT -> non empty Subset of {0,1}* means
:: BITS2: def 1
 for x being Any holds x ∈ i* iff
        ex b being Bits st x = b &
                for k being Nat st k ∈ dom b holds b.k = 0;
end;

definition
        mode UNat is Element of UNAT;
end;
```

Here, UNAT is the set of finite strings of 0's, including the empty one. "{0,1}*" is the set whose elements are Bits, so that this definition makes the UNat mode widen automatically to the Bits mode.

The correspondence between UNat and Nat modes in MIZAR is established by the following definitions.

```
definition :: U2N_def
        let u be UNat;
        func U2N u -> Nat means
:: BITS2: def 4
 it = len u;
end;

definition :: N2U_def
        let n be Nat;
        func N2U n -> UNat means
:: BITS2: def 5
 len it = n;
end;

theorem :: BITS2:16 :: UTwos
 for u being UNat for n being Nat holds U2N u = n iff u = N2U n;
```

The last theorem establishes that UNat is a "true" implementation of Nat.

4.2.1 Unary addition

Here is the definition of unary addition on UNat type.

```
definition :: Uplus_Def
        let p, q be UNat;
        func p + q -> UNat means
:: BITS2: def 6
 it = p ^ q;
end;
```

The program, however, comes from the proof of this theorem.

```
theorem :: BITS2:19 :: C_Uplus
 for a,b being UNat holds ex c being UNat st c = a + b;
```

The program uses the catenation operator. Here is the Lisp "declaration" of the extracted program.

```
(lambda (a)
  (lambda (b)
    (#R( c) #R( c = a + b) ) ) )
```

4.2.2 Unary multiplication

Finally, here is the multiplication performed on unary naturals, `UNat`. First comes the definition and then the theorem whose realization implements the multiplication functor in Lisp.

```
definition :: UNat_mult_def
        let u, v be UNat;
        func u · v -> UNat means
:: BITS2: def 7
 U2N u · U2N v = U2N it;
end;

theorem :: BITS2:22 :: C_UNat_mult
 for u being UNat holds
        for v being UNat ex uv being UNat st uv = u · v;
```

This program incorporates induction over the first argument. Here is the outline of its algorithm:

1. check if the first argument is empty, if so return 0 (`nil`);
2. strip one 0 off the first argument;
3. call itself recursively with remainder of first argument from step 2 and the second argument;
4. add the second argument to the result of the recursive call and return the value.

The following is the signature of this program.

```
(lambda (u)
  (lambda (v)
    (#R( uv) #R( uv = u · v) ) ) )
```

4.3 Binary naturals

Here is the implementation of binary naturals and the theorems whose realizations implement the four arithmetic operations. All programs implement "the classical algorithms" as in Knuth [9].

4.3.1 Definition of `BNat` mode

The `Bits` implementation of binary naturals is given in the following definitions.

```
definition :: BNAT_def
        func BNAT -> non empty Subset of {0,1}* means
:: BITS2: def 9
 for x being Any holds x ∈ it iff
        ex b being Bits st x = b & (len b = 0 or b.1 = 1);
end;

definition
        mode BNat is Element of BNAT;
end;
```

Here, BNAT is defined to be a subset of Bits such that its elements start with 1 or are empty. This ensures a one-to-one correspondence between BNat and Nat which is shown by the following theorems.

First we define MIZAR functor B2N which translates Bits to Nat.

```
definition :: B2N_Def
        let b be Bits;
        func B2N b -> Nat means
:: BITS2: def 8

        (len b = 0 implies it = 0) &
        (len b > 0 implies
            ex p being non empty FinSequence of NAT st
                it = p.(len b) & len p = len b & p.1 = b.1 &
                for n being Element of dom p st n < len b holds
                    ex bn being Nat st
                        bn = b.(n+1) & p.(n+1) = 2 · p.n + bn);
end;
```

This definition uses sequence p to store intermediate results of the translation. The following two theorems state this functor's recursive properties.

```
theorem :: BITS2:26 :: B2N_def
 B2N nil = 0 &
        (for b being Bits holds
                B2N (b ^ '0) = 2 · B2N b &
                B2N (b ^ '1) = 2 · B2N b + 1);

theorem :: BITS2:32 :: B2N_def2
 for b being Bits holds
        B2N ( nil ) = 0 &
        B2N ('0 ^ b) = B2N b &
        B2N ('1 ^ b) = 2 #N len b + B2N b;
```

The first uses tail recursion and the second one uses the "head" recursion. #N is a natural power functor.

Then comes the conversion in the opposite direction, from Nat to BNat.

```
definition :: N2B_Def
        let n be Nat;
        func N2B n -> BNat means
:: BITS2: def 12
 B2N it = n;
end;
```

And then these two theorems establish that BNat implements Nat in MIZAR.

```
theorem :: BITS2:45 :: NAT_BNAT_NAT
 for n being Nat holds B2N N2B n = n;

theorem :: BITS2:46 :: BNAT_NAT_BNAT
 for b being BNat holds N2B B2N b = b;
```

4.4 Binary addition

The addition on BNat type is defined by the following theorem

```
definition
        let p, q be Bits;
        func p + q -> BNat means
:: BITS3: def 3
 B2N it = B2N p + B2N q;
        commutativity;
end;
```

The proof of the following theorem contains the program which adds two BNat terms.

```
theorem :: BITS3:10 :: C_Bits_plus
 for a,b being Bits st len a = len b holds
            ex c being Bits st ex cu being Bit st len c = len a &
                B2N(cu ^ c) = B2N(a + b);
```

The cu term stands for a "carry unit" which contains the carry bit in performing the addition. Bit is just a narrowing of the type Bits to Bits of length 1. The proof employs the standard "column addition" algorithm which runs in $O(n)$ where n is the length of the longer of addends. The program has the following Lisp template.

```
(lambda (a)
 (lambda (b)
  (lambda ( len a = len b)
   (#R( c)
    (#R( cu)
     (#R( len c = len a & B2N(cu ^ c) = B2N(a + b)) ) ) ) ) ) )
```

The following theorem is a "wrapper" for the above one which is less restrictive on its arguments

```
theorem :: BITS3:11 :: C_BNat_plus
 for p, q being Bits ex pq being BNat st pq = p + q;
```

The Lisp signature of this program is also simpler than the previous one.

```
(lambda (p)
  (lambda (q)
    (#R( pq) #R( pq = p + q) ) ) )
```

Note that addition here is performed on `Bits` and is not restricted to the narrower type `BNat`. That is true for all arithmetic operations.

4.5 Binary subtraction

The following definition defines the "minus" functor on `BNat` terms.

```
definition
        let p, q be Bits;
        func p - q -> BNat means
:: BITS3: def 5
 (B2N p ≥ B2N q implies
                B2N it = B2N p - B2N q) &
        (B2N p < B2N q implies
                B2N it = 2 #N len q + B2N p - B2N q);
end;
```

If the minuend is less than the subtrahend then the result is presented in modulo $2^{\texttt{len q}}$. The following theorem's proof implements the program of subtraction.

```
theorem :: BITS3:15 :: C_Bits_minus
 for a,b being Bits st len a = len b holds
        ex c being Bits st ex sb being Bit st len c = len a &
            B2N(sb ^ a) = B2N(b + c) &
            (B2N a < B2N b iff sb = '1);
```

Here the `Bit` mode is just a narrowing of type from `Bits` to `Bits` of length one. Note that this theorem does not use the "minus" functor. It actually occurs before the definition of the "minus" functor and, what more, it was used in the definition of the binary subtraction. Here, `c` is the result and `sb` implements the sign bit. The proof uses the standard "column subtraction" with "borrowing" from the higher order digits. The program runs in linear order, in the length of its arguments, in number of recursive calls. Here is the Lisp signature of its realization.

```
(lambda (a)
  (lambda (b)
    (lambda ( len a = len b)
      (#R( c)
```

```
(#R( cb)
 (#R( len c = len a &
              B2N(sb ^ a) = B2N(b + c) &
              (B2N a < B2N b iff sb = '1)))))))
```

Here is the "cleanup" theorem which is more "user friendly" and uses "-" notation.

```
theorem :: BITS3:22 :: C_BNat_minus
 for p,q being Bits holds ex pq being BNat st ex s being Bit st
       pq = p - q & (B2N p < B2N q iff s = '1);
```

I left the sign bit "s" in so that if someone needs it, then it is available. The realization of the above theorem has the following Lisp signature.

```
(lambda (p)
  (lambda (q)
    (#R( pq)
     (#R( s) #R( pq = p - q & (B2N p < B2N q iff s = '1)))))))
```

4.6 Binary multiplication

The definition of multiplication and its implementation theorems are given below.

```
definition
        let p, q be Bits;
        func p · q -> BNat means
:: BITS3: def 6
 it = N2B(B2N p · B2N q);
        commutativity;
end;

theorem :: BITS3:23 :: C_BNat_t
 for a,b being Bits holds
        ex c being BNat st ex u being UNat st len u = len a & c = a · b;

theorem :: BITS3:24 :: C_BNat_times
 for a,b being Bits ex c being BNat st c = a · b;
```

Here again, the first theorem is the "work horse" of the multiplication and the second one is the interface "wrapper" to the first one. The term u holds the number of zeros by which we shift the second argument in the current step of the algorithm. Without passing u from "below" we would have to compute it at each step.

The algorithm implemented in the proof is the standard "shift and add" algorithm that is of quadratic order in length of the arguments. The Lisp signatures of the realizations of the above theorems are given below.

```
(lambda (a)
 (lambda (b)
  (#R( c)
   (#R( u) #R( len u = len a &
                          c = a · b) ) ) ) )

(lambda (a)
 (lambda (b)
  (#R( c) #R( c = a · b) ) ) )
```

4.7 Binary division

The division program is the "star" of this experiment. Division of **Bits** is defined in terms of integer division, "**div**", and modulo, "**mod**", operations on MIZAR **Nat** mode. These operations are part of the natural numbers definition described in "NAT_1" Main MIZAR Library abstract [1]. The result of the division is a pair of **BNat** terms of which the first one is the quotient and the second one is the residue of the division. Both results are **nil** when **b** is **nil** which makes this functor total. That is also the way the division of **Nat** terms in MIZAR is defined.

```
definition
        let a,b be Bits;
        func a / b -> Element of [: BNAT, BNAT :] means
:: BITS3: def 7
 it‘1 = N2B(B2N a div B2N b) &
        it‘2 = N2B(B2N a mod B2N b);
end;
```

The following theorem, whose proof implements the binary division, is the culmination of the whole experiment. It computes the quotient and the residue of binary integer division performed on **Bits**. This theorem presents a non-trivial piece of computation. Its purpose is to convince others that it is possible to write some "serious" programs and have them checked for correctness using the MIZAR and MIZ2LISP tools. After all this hype the theorem itself does not look like much.

```
theorem :: BITS3:36 :: C_BNat_div
 for a,b being BNat holds ex q,r being BNat st
        a / b = [q,r];
```

The proof utilizes the standard long division algorithm from elementary arithmetic that runs in quadratic number of recursive calls, in term of the length of the arguments. The structure of the proof is an elaborate multilevel case analysis. Thus here it is: an efficient program extracted from a proof which is derived from the axioms of set and number theories. The Lisp signature of the realization of this proof is the following:

```
(lambda (a)
  (lambda (b)
    (#R(q)
      (#R(r) #R(a / b = [q,r]) ) ) ) )
```

A list of all definitions, theorems and schemes, without proofs is attached as appendices A, B and C.

4.8 Code extraction example – binary multiplication

The purpose of this example is to illustrate how the parts of this and the previous chapter come together in extracting of Lisp programs from the constructive proofs. I present, here, the proof of the induction behind the binary multiplication. The non-constructive parts of the proof have been excluded in order to simplify and clarify the example. This example includes all of the produced Lisp realizations, which were edited to preserve space.

The inductive step is where the program comes from. This program is then processed by the induction scheme to arrange for the proper recursive calling, see section 3.3.9. The output of an XLISP session in which I ran this program concludes this section.

4.8.1 Description of the MIZAR constructs used and their realizations

The declarations of free variables with the "`let`" keyword and the declarations of assumptions with the "`assume`" keyword correspond to lambda abstractions.

The MIZAR labels are translated into local Lisp variables that have the computational content of their statements justifications assigned to them.

The "`v is Bits`" proposition takes its content from the "`v`" variable.

Each constructive scheme inference which starts with the keyword "`from`" corresponds to a programming step, which applies the Lisp code of the scheme's realization to the realizations of scheme's arguments. See section 3.3 for description of the basic constructive inferences. This proof uses a couple of compound constructive scheme inferences:

- `C_U2_E` combines 2 universal eliminations, see appendix A.3;
- `C_UI_E` is described in section 3.6.

Assigning the results of scheme applications to local variables eliminates the possibility of making redundant function calls which may result in the runtime deterioration to an exponential order.

The `reconsider` statement assigns the computational content of the term on the right to the term on the left. This is handled internally by the MIZ2LISP tool.

The `consider` statement is handled in the following way.

- First the computational content of the justification is "saved" in the "*Consider:?*" local variable, where "*?*" stands for a unique system assigned number.
- Each of the introduced variables is assigned the first element of "*Consider:?*", then this element is stripped to form the realization for processing the next variable or the resulting statement; the variables are processed internally by MIZ2LISP.

"*mfirst*" and "*mrest*" are equivalent to Lisp's "*car*" and "*cdr*".

Assumptions without a label get "*Null:?:*" realizations. It is impossible to reference these by a constructive inference.

Computational content of conclusions is assigned to the "*thus:*" local variable. This arises from the uniform handling of all constructive scheme inferences.

The description of `C_BNat_plus` theorem can be found in section 4.4. There are several external inferences. The theorems can be found in appendices. The following external references are used in the constructive inferences of this proof.

- `theorem :: BITS1:26 :: C_nil_or_not`
  `for b being Bits holds b = nil or b <> nil;`
  has the built-in function that tells if a given argument is empty string for its realization, see section 3.2.6.
- `theorem :: BITS2:8 :: C_U0`
  `U0 is Bits;`
  contains the computational content of "`U0`" which is defined as "`nil`", see appendix B.2. Its Lisp realization is the built-in basic empty string "*'()*".
- `theorem :: BITS1:29 :: C_split`
  `for p, q being Bits ex r1, r2 being Bits st p ^' q = [r1, r2];`
  has the computational content of the built-in split destructor, see section 3.2.5.
- `theorem :: BITS1:22 :: C_zero`
  `'0 is Bits;`
  has the computational content of the built-in basic string "`'0`" which is realized by "*'(0)*" in Lisp.
- `theorem :: BITS2:18 :: C_Ucat`
  `for a,b being UNat ex c being UNat st c = a ^ b;`
  is a "wrapper" for the built-in catenation, thus its functionality is the same, see appendix B.2.
- `theorem :: BITS2:9 :: C_U1`
  `U1 is Bits;`
  "`U1`" is realized by "*'(0)*", see appendix B.2.

- `theorem :: BITS1:27 :: C_unit_0_or_1`
  `for b being Bits st len b = 1 holds b = '0 or b = '1;`
  has the built-in realization of procedure that decides if a "bit" is 0 or 1, see section 3.2.7.

- `theorem :: BITS1:28 :: C_cat`
  `for p, q being Bits ex r being Bits st r = p ^ q;`
  has the built-in realization of the catenation, see section 3.2.4.

4.8.2 The program and its Lisp realization

The Lisp "translation" of each MIZAR item that has a corresponding Lisp construct is flushed to the right in the following lines. ":" stand for the parts of excluded reasoning.

```
IS: for a being Bits st                                        (setq IS:
        (for a' being Bits st len a' < len a holds
             for b being Bits holds ex c being BNat st
                 ex u being UNat st len u = len a' &
                    c = a' · b)
        holds for b being Bits holds ex c being BNat st
              ex u being UNat st len u = len a &
                 c = a · b
proof
        let a be Bits;                                         (lambda ( a )
a:      a is Bits;                                             (let ((a: a))
        assume
IH:     for a' being Bits st len a' < len a holds
            for b being Bits holds ex c being BNat st
                ex u being UNat st len u = len a' &
                   c = a' · b;                                 (lambda ( IH: )
        let b be Bits;                                         (lambda ( b )
b:      b is Bits;                                             (let ((b: b))
csa:    a = nil or a <> nil from C_U_E(BITS1:26,a);
                                  (let ((csa: (funcall (funcall c-universal-elim BITS1:26) a:)))
csa1:   now                                                    (let ((csa1:
            assume
anil:       a = nil;                                           (lambda ( anil: )
```

```
 :
            reconsider c = nil as BNat by BITS2:def 10;
c:          c is Bits;
                                                        (let ((c: c-nil))

 :
            then
p:          len U0 = len a & c = a · b by anil,BITS2:10,BNat_times_Def;
pv:         ex u being UNat st len u = len a & c = a · b
                                                  from C_E_I(p,BITS2:8);
                                    (let ((pv: (funcall (funcall c-existential-intro t) BITS2:8)))
            thus
            ex c being BNat st ex u being UNat st len u = len a & c = a · b
                                                       from C_E_I(pv,c);
                                    (let ((thus: (funcall (funcall c-existential-intro pv:) c:)))
                                                                  thus:))))))
        end;
csa2:   now
                                                                  (let ((csa2:
            assume
anil:       a <> nil;
                                                          (lambda ( anil: )
            consider af,ar being Bits such that
afar:       a ^‘ ’0 = [af,ar] from C_U2_E(BITS1:29,a,BITS1:22);
          (let ((Consider:80 (funcall (funcall (funcall C_U2_E BITS1:29) a:) BITS1:22)))
                                    (let ((afar: (funcall mrest (funcall mrest Consider:80))))
af:         af is Bits;
                                                    (let ((af: (funcall mfirst Consider:80)))
ar:         ar is Bits;
                                    (let ((ar: (funcall mfirst (funcall mrest Consider:80))))
 :
lara:       len ar < len a by anil,afar,BITS1:38;
ih0:        for b being Bits holds ex c being BNat st ex u being UNat st
                len u = len ar & c = ar · b from C_UI_E(IH,ar,lara);
                                    (let ((ih0: (funcall (funcall (funcall C_UI_E IH:) ar:) t)))
            consider arc being BNat such that
arc0:       ex u being UNat st len u = len ar & arc = ar · b
                                                       from C_U_E(ih0,b);
                                  (let ((Consider:81 (funcall (funcall c-universal-elim ih0:) b:)))
                                                (let ((arc0: (funcall mrest Consider:81))))
arc:        arc is Bits;
                                                    (let ((arc: (funcall mfirst Consider:81)))
            consider aru being UNat such that
aru0:       len aru = len ar & arc = ar · b from C_E_E(arc0);
                                    (let ((Consider:82 (funcall c-existential-elim arc0:)))
                                                (let ((aru0: (funcall mrest Consider:82)))
```

```
 :
aru:        aru is Bits;
                                        (let ((aru: (funcall mfirst Consider:82)))
            consider u being UNat such that
u0:         u = U1 ^ aru from C_U2_E(BITS2:18,BITS2:9,aru);
       (let ((Consider:83 (funcall (funcall (funcall C_U2_E BITS2:18) BITS2:9) aru:)))
                                        (let ((u0: (funcall mrest Consider:83)))
u:          u is Bits;
                                        (let ((u: (funcall mfirst Consider:83)))
 :
laf:        len af = 1 by anil,afar,BITS1:38;
 :
csaf:       af = '0 or af = '1 from C_UI_E(BITS1:27,af,laf);
                    (let ((csaf: (funcall (funcall (funcall C_UI_E BITS1:27) af:) t)))
csaf1:      now
                                                                (let ((csaf1:
                assume
                af = '0;
                                                        (lambda ( Null:84: )
 :
                then
p:              len u = len a & arc = a · b by lula,BNat_times_Def;
pv:             ex u being UNat st len u = len a & arc = a · b
                                                          from C_E_I(p,u);
                                    (let ((pv: (funcall (funcall c-existential-intro t) u:)))
                thus
                ex c being BNat st ex u being UNat st len u = len a &
                                          c = a · b from C_E_I(pv,arc);
                                (let ((thus: (funcall (funcall c-existential-intro pv:) arc:)))
                                                                thus:)))))
            end;
csaf2:      now
                                                                (let ((csaf2:
                assume
af1:            af = '1;
                                                        (lambda ( af1: )
 :
                consider bu being Bits such that
bu0:            bu = b ^ aru from C_U2_E(BITS1:28,b,aru);
              (let ((Consider:85 (funcall (funcall (funcall C_U2_E BITS1:28) b:) aru:)))
                                        (let ((bu0: (funcall mrest Consider:85)))
bu:             bu is Bits;
                                        (let ((bu: (funcall mfirst Consider:85)))
 :
```

```
                consider c being BNat such that
c0:             c = bu + arc from C_U2_E(C_BNat_plus,bu,arc);
        (let ((Consider:86 (funcall (funcall (funcall C_U2_E C_BNat_plus:) bu:) arc:)))
                                                    (let ((c0: (funcall mrest Consider:86)))
c:              c is Bits;
                                                    (let ((c: (funcall mfirst Consider:86)))
 :
                then
p:              len u = len a & c = a · b by lula,BNat_times_Def;
pv:             ex u being UNat st len u = len a & c = a · b
                                                        from C_E_I(p,u);
                                            (let ((pv: (funcall (funcall c-existential-intro t) u:)))
                thus
                ex c being BNat st ex u being UNat st len u = len a &
                                                        c = a · b from C_E_I(pv,c);
                                        (let ((thus: (funcall (funcall c-existential-intro pv:) c:)))
                                                                        thus:))))))))))

            end;
            thus
            ex c being BNat st ex u being UNat st len u = len a & c = a · b
                                                    from C_O_E(csaf,csaf1,csaf2);
                    (let ((thus: (funcall (funcall (funcall c-or-elim csaf:) csaf1:) csaf2:)))
                                                                    thus:))))))))))))))))))))

        end;
        thus
        ex c being BNat st ex u being UNat st len u = len a & c = a · b
                                                        from C_O_E(csa,csa1,csa2);
                    (let ((thus: (funcall (funcall (funcall c-or-elim csa:) csa1:) csa2:)))
                                                                    thus:))))))))))

end;

theorem :: C_BNat_t
C_BNat_t: for a,b being Bits holds
        ex c being BNat st ex u being UNat st len u = len a & c = a · b
                                                            from C_Comp_Ind(IS);
                                                (setq C_BNat_t: (funcall c-induction IS:))
```

4.8.3 Invoking the Lisp program

Here are two sample runs of the multiplication program that resulted from the above proof. The first bit string is the result of the multiplication and the second string of zeroes represents the "shift" in the next step of the program, if there was one. "*c-ind-count*" counts the number of inductive/recursive calls by the induction scheme program, see section 3.3.9.

First we calculate "$42 \times 24 = 1008$".

```
>(setq c-ind-count 0)
0
>(funcall (funcall c_bnat_t: '(1 0 1 0 1 0)) '(1 1 0 0 0))
((1 1 1 1 1 1 0 0 0 0) ((0 0 0 0 0 0) T))
>(print c-ind-count)
57
57
```

And then we calculate "$1578 \times 1368 = 2158704$".

```
>(setq c-ind-count 0)
0
>(funcall (funcall c_bnat_t: '(1 1 0 0 0 1 0 1 0 1 0)) '(1 0 1 0 1 0 1 1 0 0 0))
((1 0 0 0 0 0 1 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0) ((0 0 0 0 0 0 0 0 0 0 0 0) T))
>(print c-ind-count)
190
190
```

After I analyzed the proofs of the multiplication and the addition and their corresponding Lisp code realizations, I came to conclusion that the binary multiplication program should perform no more than the order of $O(m \times n)$ inductive/recursive calls, where m and n are the lengths of the first and the second arguments. Each addition should perform a linear order number of inductive calls in terms of the length of its arguments, and each argument is not longer than $m + n$. Note that there are at most m calls to the addition function in the multiplication program. The above results seem to follow my findings.

Chapter 5

Implementation notes

MIZ2LISP is the software that translates constructive parts of a MIZAR article into Lisp code. It is implemented as one program. It reads in the MIZAR article and outputs its Lisp translation into another file. It also reads in a file that specifies which MIZAR items get the built-in computational contents as well as the files that specify computational contents of theorems and schemes of other articles that have been already translated by MIZ2LISP.

I must say that the implementation of the MIZ2LISP translator took more effort than all the other parts of this experiment combined. There are three parts to the MIZ2LISP translation: parsing the MIZAR article, translating it, and handling external files.

I did not test my parser on many MIZAR articles. I was more concerned with parsing just my articles, since the goal of the experiment was not to parse every MIZAR article but to extract programs from some of them. Thus I was satisfied with MIZ2LISP parsing abilities when it processed my articles correctly. I do not see the parsing capabilities of MIZ2LISP as an impediment to the project since I was still able to extract programs from proofs checked by the PC MIZAR processor.

5.1 Parsing of the MIZAR language

The MIZAR parser took by far the most time to implement. The MIZAR language is rich in features that are context-sensitive and there is no comprehensive documentation for it. The only complete description of the grammar are the source codes themselves and in these the grammar is already written in the form of parsing tables. Also, the MIZAR language is a research experiment in itself. Although it has been quite stable for the past few years, there is "talk" about some dramatic changes ahead. In my experiment I used PC MIZAR version 5.1.03.

To tackle the MIZAR grammar I used the *Bison* and *Flex* compiler tools. The *Bison* description still has some "shift/reduce" conflicts but they do not seem to cause any trouble at this time. The *Bison* and *Flex* combination allowed me to parse some of the context-sensitive constructs of the MIZAR language. In particular I found

the context switching mechanism of *Flex* very helpful.

For my guide into the MIZAR language I used some outdated grammar diagrams from a few years ago and the MIZAR articles themselves. The whole process was done in a trial and error fashion.

The articles that were both checked and successfully translated into Lisp were a cross section of the MIZ2LISP compiler grammar and the MIZAR language. The exact subset of the MIZAR language that is accepted by the compiler extension is not well known even by me. One feature which I know is not accepted by the parser is MIZAR's "iterative equality". However, most of the MIZAR constructs are parsed and handled properly.

There is one feature of the language that is especially tricky to handle without dynamically altering parsing tables. Users in MIZAR can introduce their own symbols in the vocabulary files and they could be almost any sequence of printable characters and even some non-ascii ones. MIZAR allows for symbols, like "`Function-like`", with dashes inside. To get around parsing these symbols I "cheated" and included them in the MIZ2LISP grammar. Most of them are usually very common symbols like "`=`" or "`+`" and appear in the "hidden.voc" vocabulary file that is attached to all articles by the PC MIZAR system.

5.2 Processing the built-in base of computational contents

MIZ2LISP assigns predefined built-in computational contents to the computational primitives in MIZAR articles. The computational contents of the bit strings model and the basic constructive inferences are loaded before the MIZAR article is read. Then, during parsing, when a statement for which realization already exists is detected that statement receives that realization.

This is a very flexible way of assigning computational content to MIZAR statements. One can easily change the built-in realization and the statements that get them without even touching the MIZ2LISP software. One only has to design the realizations of inferences so that they reflect how they are being applied by MIZ2LISP.

What item of MIZAR text receives what realization is specified in a special file that is read during the initialization stage. The links in this file have the following structure:

1. the name of the MIZAR item which will receive the realization;
2. the type of the MIZAR item; right now only labels, symbols and schemes can receive predefined realizations;
3. the name of a Lisp expression that constitutes the realization.

The items of MIZAR text that receive the built-in realization are the ones listed in sections 3.2 and 3.3. The items listed in section 3.4 are handled through MIZ2LISP directly in its actions of *Bison* grammar specification.

5.3 Passing information around – building libraries

There are two reasons for splitting MIZAR articles into several smaller ones. One is to group the theorems into topical articles to build a library. Another, more pragmatic reason, is that large articles overflow the PC MIZAR processor's memory resources. So, only small enough articles can be processed by the PC MIZAR system. As its name indicates, the PC MIZAR system is implemented on an IBM compatible PC running the DOS operating system. Because of the limitations DOS imposes on programs, the size of the input articles has to be kept relatively small so that PC MIZAR programs do not run out of memory.

In this light, I was forced to split my theorems into 3 MIZAR articles. This introduced another issue to be tangled with in the experiment; namely, how to pass information about the computational contents among articles. Luckily, the flexibility of handling most of the computational contents orthogonally to the compiling process paid off at these crossroads.

There are three types of statements that can be exported for the use in other MIZAR articles. They are "definitional statements", "definitions of schemes" and "theorems". The definitional statements, at this time, cannot have any content assigned to them. The links to the other two types of statements that have computational content are saved in two special files, one for the schemes and the other for the theorems. These files in turn are read by the translator when another article asks to import schemes and theorems of another MIZAR article. This takes place in the "environment" section of the MIZAR article. The result is that the references to imported items, effectively having predefined computational content, have their realization assigned to them. These contents are the names of already extracted Lisp programs from imported articles.

All MIZAR proper text items that have realizations produced for them are given their respective labels or scheme names as reference points in the generated Lisp code. These references then become their realizations that are written to the "theorems" and "schemes" files for use by other articles.

5.4 Running compiled Lisp code

The proofs from MIZAR articles are compiled into a Lisp variant, XLISP. After the article has been checked as correct by the PC MIZAR system and then compiled into the Lisp code, it can be loaded into the XLISP interpreter. First, the library of built-in bases of computational contents has to be loaded. Then, the Lisp files output by the MIZ2LISP compiler should be loaded in the order of their dependencies on each other. Otherwise XLISP will complain about undefined terms.

The programs themselves are lambda expressions and are invoked by applying these lambda expression to the arguments on which the computation is to be performed.

5.4.1 Efficiency considerations

I did not concern myself much with the efficiency of the generated Lisp code. The correctness of the programs was the primary issue in this experiment. But when a program that was expected to run in linear time, ran in exponential time, then I got concerned.

In my first attempt at generating Lisp code I simply expanded the components of all computational terms. This expansion terminated on the text proper statements of MIZAR. Everything was fine until I ran the program that implemented binary addition. In binary addition there are two terms computed: the result and the "carry" bit. When these two terms were expanded there were two recursive calls in the program instead of one. To fix this problem I had to save the results of the first recursive call and use them where the second call was made. I solved this problem by assigning every step's computation to a local variable. I agree that it is an "over kill" but as I already mentioned "small" grain efficiency is not my concern here. In the end only one recursive call is made, no matter how many objects it computes.

5.5 Unfinished Business

Although, I feel that I was able to answer the core questions concerned with translating formal proofs in MIZAR language into executable programs, there are minor practical aspects that are not attended to in my implementation. Here I outline them so that these problems can be taken care of at the later time and make anyone interested aware of them. I think that they present a very good warm up exercise for someone who would like to continue this project along the lines proposed in chapter 6. They also present a good starting point into the problems of parsing the MIZAR language.

5.5.1 Access to MIZAR library

One of the unfinished parts of the project is the interface to the MIZAR library where many of the symbols I used are introduced. For the compilation process to be fully rooted in the derivations of the PC MIZAR system, descriptions of symbols used in the translated articles, but defined in imported ones, must be fetched from the library.

As it sits right now, I manually created a "cheat" file that contains descriptions of the symbols used in the articles. The dynamic introduction of symbols that use characters other than alphanumeric may be especially tricky since they do not fit nicely under one group of regular expressions. Right now I included them in the scanner specification.

One way to solve this problem is to check all input against introduced symbols before the rest of the scanner gets it. Another might be some sort of preprocessor stage that generates the scanner which is then linked dynamically into the MIZ2LISP program. As it is at the moment, MIZ2LISP's translation definitely falls short of being a fully automated compilation process.

5.5.2 Renaming references

In MIZAR , when I make a local reference to a previous result within the same article, I use the labels I assigned to the referenced statement. However, once the article is made a part of a library, PC MIZAR assigns numbers to the theorems contained in it. Other articles have to use these numbers to reference the theorems. Fortunately, the schemes retain their names. The problem, now, is to connect these numbers with the names of the extracted Lisp programs. It seems that the numbers assigned to the theorems are in order of appearance starting with 1; thus, MIZ2LISP is capable of reproducing this sequence and output this correspondence in the theorems file.

As mentioned above, in the compile run two other files are created that contain the references to the programs behind the theorems and schemes. I did not take care of the names of these files yet and they have to be changed manually so that they reflect the names used in the "environment" section of the MIZAR article. Also the references in the theorems file have to be renamed. It is only done once; nonetheless, it should be automated by the MIZ2LISP software.

5.5.3 Dealing with case insensitivity of XLISP

More significant and harder to find inconsistencies may occur because of the mono-case of symbols in XLISP. The MIZAR language is case-sensitive so the labels "a:" and "A:" are different. Unfortunately, this difference is lost inside XLISP and one label can shadow another inside Lisp without a warning. I would like to blame it on the shortcomings of XLISP, but it was a poor choice on my part to use it for my implementation.

5.5.4 Fitness of MIZ2LISP software

I was able to use this implementation of MIZ2LISP for my experiments and was successful at extracting Lisp programs from MIZAR proofs. However, MIZ2LISP is an experiment in progress, and should not be viewed as anything more. The goal was never to develop a production quality system.

Chapter 6

Unresolved issues and future directions

Here I would like to outline several issues that I find important but which still remain unresolved. The most significant one is that although the results of the programs are proven to be correct there is still the possibility of a run-time failure. This is because at this time no checking is implemented to verify that whenever a computational content is expected one is provided.

There are also more general issues connected with using the extracted programs outside of the formal environment. At this time there is no guarantee as to what the extracted programs will do when given wrong arguments. They will most likely crash.

On the more practical side, there is still much room for improvement left in the MIZ2LISP translator itself. These improvements mostly deal with the shortfalls already mentioned in chapter 5. Although, it may not be a research project in itself, finishing some of the "unfinished" aspects of MIZ2LISP could constitute a project of a practical value.

Also, more experimentation with more complex problems than arithmetic on binary naturals would give us a better idea about how much effort is needed in order to deal with complex issues. Thanks to the flexibility and extensibility of MIZ2LISP one could easily experiment with different basic computational models.

6.1 Avoiding runtime errors

MIZ2LISP does not guarantee that the produced programs will run at all. The program given in the example from section 3.5.1 will crash when "t" is applied to an argument because `C_Comp_Ind` expects a lambda expression. This is because there is no checking for the possibility of a lambda expression which expects an argument with a content being applied to an item which does not have a computational content and is only realized by "t". This I see as the next step on the road to extracting correct programs from constructive proofs. Below I present my thoughts on how this

runnability issue could be resolved.

6.1.1 Brute force

One solution is to include tracing information in the produced code itself. This way one could test the program by trying to execute every statement in the program at least once. If that happens without any crashes then we have a program that will never crash. The reasoning behind this approach is that "empty realizations" are static objects introduced during the compile time and do not depend on any dynamic properties of the program. I should add that the following assumptions about the system are made:

- the verification step guarantees that the types of the realizations are correct, that is, the types of arguments and formal parameters match; thus, either there is a correct computation supplied or none at all;
- when an empty realization, "t", is used where a "true" computation is expected, the extracted program will crash;
- "true" computations cannot result in "t".

Thus if empty realizations do occur in places where a "true" computational content is expected, they will cause the program to fail. And if they do not occur in the "wrong" places then the arguments came from a proper computational content and thus they are guaranteed to always have appropriate shape.

This is a solution that would be easy to implement with only slight additions to the current version of MIZ2LISP. Since it only requires a relatively small set of input cases to be checked, the approach is quite practical. This amounts to a static check, only approached from a less organized angle.

6.1.2 Extracting programs without runtime errors

What I view as the "true" solution to the "runnability" problem is to check statically, at the compile time or perhaps as an additional step, for possible invocation of lambda expressions which use the arguments that have empty realizations of "t" in their computations.

I propose a resolution that goes along this line: "if the problem is hard to solve then change the problem". In other words I propose a restriction on how the constructive inferences can be made which will simplify the problem. My restriction is that there be no discharging of implications which use their antecedents in constructive derivations of their conclusions. That is, all antecedents are only used as "guards" and should not carry any computational content. The only exception would be the assumption of inductive hypothesis which is required in order to make the recursive call inside induction.

How would this work? First, mark the formal arguments of all the elementary constructive inference rules to denote whether they require a "true" computational content in its place:

- C_O_I_l (N),
- C_O_I_r (N),
- C_O_E (C, C, C),
- C_I_E (C, N),
- C_U_E (C, C),
- C_E_I (N, C),
- C_E_E (C),
- C_EQ (N, C),
- C_Comp_Ind (C),

where "C" means that in its place a reference with a non empty realization is required in order for the inference to produce realization with content for the inferred proposition. Arguments in the "N" positions may or may not have realizations.

Now, all lambda bound variables in lambda expressions are marked with "C" if somewhere within its scope it is being used in a constructive inference in a "C" position. This means that we used them in a constructive way.

And here comes the tricky part. The restriction is enforced when an inference using an implication is applied. The are only two places where discharging of implication occurs, inside the C_I_E and C_O_E schemes. Then when processing these two inferences the realizations of the implications are checked and if their lambda bound variables are marked with "C", then the constructive inference is rejected and no realization is produced for the proposition.

The variable assumptions are assumed to always require realizations with computational content and this is reflected in markings of the "C_U_E" scheme.

So, now we are left with only one place where a realization with computational content is expected by an implication, namely, the inductive hypothesis. However, the argument to that lambda expression is prepared by the induction scheme which is guaranteed to provide a proper lambda expression for use as an inductive hypothesis.

What is achieved by the above method is that there will be no computational content produced by the constructive scheme inference unless the scheme inference is guaranteed to always receive a computational content for the parameters that require it. Since the correctness of each computational content is guaranteed by the verification stage, then only the programs with no runtime errors will be produced. There are still some details to be worked out but I leave it to the future implementation.

The above restriction forbids the passing of procedures as arguments to functions. It does not limit the functionality of the original system. Every function that takes a procedural argument can be rewritten into several functions: one function for each different procedure that the original function is called with. These functions call their respective procedures directly from within. This "rewrite" eliminates the need for the procedural parameter altogether. It does, however, require the user to prove more theorems since the "template" proofs will have to be "expanded".

6.2 Problem of using programs outside the controlled environment

As long as we are inside the controlled environment of formal reasoning, we have a guarantee that everything is being used within their specifications. The problem arises when we leave that controlled environment and try to invoke the program with arguments that are not checked for validity since there is nothing to check them. This occurs even in my experiment since I had no means of invoking programs from within PC MIZAR system. I took the produced programs and ran them inside the XLISP interpreter, which is foreign to the formal system that checked the programs.

At this point I have no other solution but to try to describe the complex outside world in the PC MIZAR language. This approach at this time, if at all feasible, is not very practical. Thus the best that one can hope for is that the correct programs will be wrapped in another programs that will check the validity of arguments. Unfortunately, this approach compromises the formal verification of the original. This is only one aspect of a larger problem of when the formal verification stops and trust, achieved through other means, in the correctness of programs begins.

Chapter 7

Summary and conclusions

Overall, I see this experiment as a success. I managed to translate all of the proofs I wanted into Lisp programs and they all ran within a reasonable time. Thus, I conclude that it is possible to use the PC MIZAR system as a verifier of the correctness of programs. I hope that the lure of the knowledge already collected in the Main MIZAR Library will be enough to ensure this project's continuation in one form or another.

I leave this experiment with a usable, although unfinished, system for conducting further experiments in the compilation of formal proofs into runnable code. I led the way in opening the MIZAR system to other experiments with constructive proofs and program extraction. Further more, I proposed a solution to the runnability problem, which stands in the way of producing "totally" correct programs.

My overall experience has been that using formal methods at such an extremely detailed level is a very laborious process and that much work is still left to be done in this area before we see any significant changes. However, I found that the effort spent on proving the correctness of specifications was paid back when I ran the resulting Lisp programs. They worked exactly as they were supposed from the very first time. This experience pleasantly surprised me as I was expecting some errors to occur even though I had earlier proved the correctness of these programs.

7.1 The future of the program extraction from constructive proofs

I started this project with great expectations regarding this approach to program correctness, but I leave it somewhat confused. Mainly, because there are still many issues unresolved by this method. It is true that this method, based on constructive proofs, offers a solution to the problem of producing correct programs. But so does the external approach mentioned in section 1.1.1.

I do not see a big difference between proving the properties of explicitly written programs and proving "just" the specifications. The "hard" work is the same, proving a lot of mathematical facts about the resulting objects. Writing programs is the "easy

part". True, there is more work involved in describing the programming environment, but the rewards of this work are twofold: one is that we have the environment in which to talk about the programs directly; and the other one is that we know that the environment is sound.

I do not know exactly how much effort would be involved in developing the programming environment for the bit strings computational model, but I suspect that, given the model's simplicity, it would be considerably less than the effort I spent on constructively proving the binary arithmetic operations.

The problem of translating the program into an implementation language is the same if not easier in the external approach since the programming steps are stated explicitly and there is no "runnability" issues involved. In other words the external approach resolves all of the issues connected with program extraction and gives us the ability to talk about the programs. The external approach seems to present a better solution to the problem of obtaining correct programs.

Bibliography

[1] G. Bancerek. The fundamental properties of natural numbers. *Formalized Mathematics*, 1(1):41–46, 1990.

[2] G. Bancerek and P. Rudnicki. Two programs for SCM. Part I — Preliminaries. PartII — Programs. *Formalized Mathematics*, 4(1):69–75, 1993.

[3] Ewa Bonarska. *An Introduction to* PC MIZAR. MIZAR User Group. Fondation Philippe le Hodey, Brussels, Av. F. Roosevelt 33 (B), 1050 Brussels, Belgium, 1990.

[4] Thierry Coquand. On the analogy between propositions and types. In Gerard Huet, editor, *Logical Foundations of Functional Programming*, The UT Year of Programming Series, pages 399–418, Reading, Massachusetts, 1990. Addison-Wesley.

[5] Edward Ho. Implementing binary trees in MIZAR-C. Master's thesis, University of Alberta, 1996.

[6] Kathleen Kippen. Implementing bit data structure in MIZAR-C. Master's thesis, University of Alberta, 1995.

[7] S.C. Kleene. *Introduction to Metatmathematics*. Van Nostrand, 1952.

[8] B. Knight. Safe strict evaluation of redundancy-free programs from proofs. Master's thesis, University of Victoria, 1994.

[9] Donald E. Knuth. *The Art of Computer Programming, Volume 2 / Seminumerical Algorithms*, pages 250–268. Addison Wesley, 2 edition, 1969.

[10] P. Martin-Lof. Constructive mathematics and computer programming. In C.A.R. Hoare and J.C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184. Prentice/Hall, 1985.

[11] Roman Matuszewski, editor. Formalized mathematics. Fondation Philippe le Hodey, Av. F. Roosevelt 33 (B), 1050 Brussels, Belgium.

[12] J.S. Moore et al. Special issue on system verification. *Journal of Automated Reasoning*, 5:409–530, apr 1989.

[13] Michał Muzalewski. *An Outline of* PC MIZAR. MIZAR User Group. Fondation Philippe le Hodey, Brussels, Av. F. Roosevelt 33 (B), 1050 Brussels, Belgium, 1993.

[14] A. Trybulec. Built-in concepts. *Formalized Mathematics*, 1(1):13–16, 1990.

[15] A. Trybulec. Tarski Grothendieck set theory. *Formalized Mathematics*, 1(1):9–11, 1990.

[16] A. Trybulec and Y. Nakamura. Some remarks on the simple concrete model of computer. *Formalized Mathematics*, 4:51–56, 1993.

[17] Andrzej Trybulec. *Some Features of the* MIZAR *Language*. Institute of Mathematics, Warsaw University in Białystok, 15-267 Białystok, Akademicka 2, Poland, 1990.

[18] A. Walenstein, J. H. Hoover, and P. Rudnicki. Programming with constructive proof in the MIZAR-C proof environment. Technical Report 92-12, University of Alberta, 1992.

Appendices

The following appendices contain abstracts, produced by the PC MIZAR system, of the MIZAR articles, from which the Lisp programs were extracted. Appendix A presents the formalization of the bit strings model, `Bits`, and the specifications of the constructive inference rules. The theorems about the unary encoded natural numbers, `UNat`, and the formalization of the binary representation of naturals, `BNat`, are stated in appendix B. Appendix C contains the constructive theorems of the four arithmetic operations on the binary naturals: addition, subtraction, multiplication and division.

The full texts of these MIZAR articles and the programs extracted by MIZ2LISP from the proofs contained within, as well as the sources for the MIZ2LISP processor, will be made available from the Mizar Web Library at the Department of Computing Science, University of Alberta. This library is currently under construction. Please, contact P. Rudnicki (*piotr@cs.ualberta.ca*) or J. Hoover (*hoover@cs.ualberta.ca*).

Appendix A

Definition of bases with computational content

BITS1.ABS

January 28, 1996

```
environ
        vocabulary
        BOOLE, FUNC_REL, SUB_OP, REAL_1, NAT_1, INT_1, ANAL,
        FINSEQ, COORD, FUNC, FINITER2, TUPLES, PI, SIGMA, NEWTON,
        POWER, POWER1, NISHIYA, BITS;
        constructors
        BOOLE, FUNCT_2, SEQ_1;
        signature
        TARSKI, BOOLE, REAL_1, SUBSET_1, NAT_1, INT_1, INT_2, FUNCT_1,
        FINSEQ_1, MCART_1, DOMAIN_1, FINSEQ_2, FINSOP_1, PREPOWER, POWER,
        BINARITH;
        theorems
        TARSKI, AXIOMS, BOOLE, REAL_1, REAL_2, SUBSET_1, NAT_1, FUNCT_1,
        FINSEQ_1, FINSEQ_2, INT_1, INT_2, MCART_1, NEWTON, PREPOWER, POWER,
        SQUARE_1, BINARITH;
        schemes
        BOOLE, NAT_1, FINSEQ_1, RECDEF_1;
        definitions
        TARSKI;
        clusters
        FUNCT_1, FINSEQ_1,FINSEQ_2;
begin
```

A.1 Preliminaries

```
theorem :: BITS1:1 :: realjunk0
```

```
0 <> 1 &
        0 < 1 &
        0 ≤ 1 &
        0 = -0 &
        1 + -1 = 0 &
        1 - 1 = 0 &
        1 = -(-1) &
        -0 > -1 &
        0 > -1 &
        0 + 1 = 1 &
        0 + -1 = -1 &
        0 - 1 = -1 &
        1 > -1 &
        1 <> -1 &
        1 ≥ -1;

theorem :: BITS1:2 :: realjunk1
 for r being Real holds
        r + 0 = r & :: 1:4
        r · 1 = r & :: 1:7
        1 · r = r & :: 1:7
        r + -r = 0 & :: 1:def 1
        (r <> 0 implies r · r" = 1) & :: 1:def 2
        -(-r) = r & :: 1:18
        0 - r = -r & :: 1:19
        r · 0 = 0 & :: 1:20
        (r <> 0 iff -r <> 0) & :: 1:22
        r - 0 = r & :: 1:25
        (r <> 0 implies r" <> 0) & :: 1:31
        (r <> 0 implies r"" = r) & :: 1:32
        (r <> 0 implies (1 / r = r" & 1 / r" = r)) & :: 1:33
        r - r = 0 & :: 1:36
        (r <> 0 implies r / r = 1) & :: 1:37
        (r < 0 iff 0 < -r) & :: 1:66
        (0 < r implies 0 < r") & :: 1:72
        0 ≤ r · r & :: 1:93
        (ex y being Real st r<y) & :: REAL_1:76
        (ex y being Real st y<r) & :: REAL_1:77

        r - 1 = r + -1 &
        r < r + 1 &
        r ≤ r + 1 &
        r <> r + 1 &
        r > r - 1 &
        r ≥ r - 1 &
        r <> r - 1 &
```

```
        (r = 0 implies r <> 1) &
        (r = 1 implies r <> 0);

theorem :: BITS1:3 :: realjunk2
 for x, y being Real holds
        x-y=x+ -y & :: REAL_1:14
        (x/y=x·y" & x/y=y"·x) & :: REAL_1:16
        (x·y=0 iff (x=0 or y=0)) & :: REAL_1:23
        (x<>0 & y<>0 implies x"·y"=(x·y)") & :: REAL_1:24
        (y<>0 implies (-x/y=(-x)/y & x/(-y)=-x/y)) & :: REAL_1:39
        (y<>0 implies x/y·y=x) & :: REAL_1:43
        (ex z being Real st (x=y+z & x=z+y)) & :: REAL_1:44
        (y<>0 implies ex z being Real st (x=y·z & x=z·y)) & :: REAL_1:45
        (x≤y iff -y≤-x) & :: REAL_1:50
        (x<y iff x≤y & x<>y) & :: REAL_1:def 5
        (x<>y implies x<y or y<x) & :: REAL_1:61
        (0<x implies y<y+x) & :: REAL_1:69
        (x<y implies ex z being Real st x<z & z<y) & :: REAL_1:75
        (x<>0 & y <> 0 implies (x/y)"=y/x) & :: REAL_1:81
        -(x-y)=y-x;

theorem :: BITS1:4 :: realjunk2x
 for r1, r2 being Real holds
        (r1 > r2 iff r1 - r2 > 0) &
        (r1 > r2 iff r2 - r1 < 0) &
        (r1 ≥ r2 iff r1 - r2 ≥ 0) &
        (r1 ≥ r2 iff r2 - r1 ≤ 0) &
        (r1 = r2 iff r1 - r2 = 0) &
        (r1 = r2 iff r2 - r1 = 0) &
        (r2 > 0 iff r1 - r2 < r1) &
        (r2 = 0 iff r1 + r2 = r1) &
        (r2 = 0 iff r2 + r1 = r1) &
        (r1 ≤ r2-1 implies r1 < r2) &
        (r1 < r2 iff r1 ≤ r2 & r1 <> r2) &
        (r1 ≤ r2 iff r1 + 1 ≤ r2 + 1) &
        (r1 ≤ r2 iff r1 - 1 ≤ r2 - 1) &
        (r1 <> r2 iff r1 + 1 <> r2 + 1) &
        (r1 <> r2 iff r1 - 1 <> r2 - 1) &
        (r1 < r2 iff r1 + 1 < r2 + 1) &
        (r1 < r2 iff r1 - 1 < r2 - 1) &
        (r1 + 1 ≤ r2 iff r1 ≤ r2 - 1) &
        (r1 + 1 < r2 iff r1 < r2 - 1) &
        (r1 + 1 <> r2 iff r1 <> r2 - 1) &
        (r1 - 1 ≤ r2 iff r1 ≤ r2 + 1) &
        (r1 - 1 < r2 iff r1 < r2 + 1) &
```

```
        (r1 - 1 <> r2 iff r1 <> r2 + 1) &
        r1 + r2 = r2 + r1 &
        r1 + (-r2) = -r2 + r1 &
        r1 - r2 = -r2 + r1;

theorem :: BITS1:5 :: realjunk3
 for x, y, z being Real holds
        ((x+y)·z=x·z + y·z & z·(x+y)=z·x + z·y) & :: REAL_1:8
        ((z<>0 & x<>y) implies
                        (x·z<>y·z & z·x<>y·z & z·x<>z·y &
        x· z<>z·y)) & :: REAL_1:9
        ((z + x = z + y or x + z = y + z or z +x = y + z or x + z = z + y)
                                                implies x=y) & :: REAL_1:10
        (x<>y iff x+z<>y+z) & :: REAL_1:11
        ((z<>0 & (x·z=y·z or z·x=z·y or
                  x·z=z·y or z·x=y·z)) implies x=y) & :: REAL_1:12
        x+y-z=x+(y-z) & :: REAL_1:17
        ((-x)·y = -(x·y) & x·(-y)=-(x·y) & (-x)·y=x·(-y)) &
                                                                    ::
REAL_1:21
        x-(y+z)=x-y-z & :: REAL_1:27
        x-(y-z)=x-y+z & :: REAL_1:28
        (x·(y-z)=x·y - x·z & (y-z)·x=y·x - z·x) &
                                                                    ::
REAL_1:29
        (x+z=y implies (x=y-z & z=y-x)) & :: REAL_1:30
        (y<>0 & z<>0 implies x/y=(x·z)/(y·z)) & :: REAL_1:38
        (z<>0 implies ( x/z + y/z = (x+y)/z ) & ( x/z - y/z = (x-y)/z )) &
                                                                    ::
REAL_1:40
        (y<>0 & z<>0 implies x/(y/z)=(x·z)/y) & :: REAL_1:42
        (x ≤ y implies (x + z ≤ y + z & x - z ≤ y - z)) &
                                                                    ::
REAL_1:49
        (x ≤ y & 0 ≤ z implies (x·z ≤y·z & z·x ≤z·y &
                z·x ≤y·z & x·z≤z·y )) & :: REAL_1:51
        (x≤y & z≤0 implies (y·z≤x·z & z·y≤z·x &
                y·z≤z· x & z·y≤x·z)) & :: REAL_1:52
        (x ≤ y iff x+z≤y+z) & :: REAL_1:53
        (x ≤ y iff x-z≤y-z) & :: REAL_1:54
        (((x≤y & y<z) or (x<y & y≤z) or (x<y & y<z)) implies x<z) &
                                                                    ::
REAL_1:58
        (x<y implies (x+z<y+z & x-z<y-z & z+x<z+y & x+z<z+y & z+x<y+z)) &
                                                                    ::
REAL_1:59
```

```
        ((x+z<y+z or z+x<z+y or x+z<z+y or z+x<y+z or x-z<y-z ) implies
x<y) &                                                              ::

REAL_1:60
        (0<z & x<y implies (x·z<y·z & z·x<z·y & x·z<z·y &
                                        z·x<y· z)) & :: REAL_1:70
        (z<0 & x<y implies (y·z<x·z & z·y<z·x & y·z<z·x &
                                        z·y<x ·z)) & :: REAL_1:71
        (0<z implies (x<y iff x/z<y/z)) & :: REAL_1:73
        (z<0 implies (x<y iff y/z<x/z)) & :: REAL_1:74
        (x+y ≤ z iff x ≤ z-y) & :: REAL_1:84
        (x+y ≤ z iff y ≤ z-x) & :: REAL_1:85
        (x ≤ y+z iff x-y ≤ z) & :: REAL_1:86
        (x ≤ y+z iff x-z ≤ y);

theorem :: BITS1:6 :: realjunk4
 for x, y, z, t being Real holds
        ((y<>0 & t<>0) implies (x/y) · (z/t) =(x·z)/(y·t)) &
                                                                    ::

REAL_1:35
        (y<>0 & t<>0 implies ( x/y + z/t =(x·t + z·y)/(y·t) )
        & ( x/y - z/t =(x·t - z·y)/(y·t) )) & :: REAL_1:41
        ((x≤y & z≤t) implies
                        (x+z≤y+t & x+z≤t+y & z+x≤t+y & z+x≤y+t)) &
                                                                    ::

REAL_1:55
        (((x<y & z≤t) or (x≤y & z<t) or (x<y & z<t)) implies
                                (x+z<y+t & z+x<y+t & z+x<t+y & x+z<t+y)) &
                                                                    ::

REAL_1:67
        (y<>0 & z<>0 & t<>0 implies (x/y)/(z/t)=(x·t)/(y·z)) &
                                                                    ::

REAL_1:82
        (((x ≤ y & z ≤ t) implies x - t ≤ y - z) &
                (((x < y & z ≤ t) or (x ≤ y & z < t) or (x < y & z < t))
                                        implies x-t < y -z));


 theorem :: BITS1:7 :: natjunk1
  for k being Nat holds
        0 ≤ k & :: NAT_1:18
        (0 <> k implies 0 < k) & :: NAT_1:19
        0 <> k + 1 & :: NAT_1:21
        (k = 0 or ex n being Nat st k = n + 1);

 theorem :: BITS1:8 :: natjunk2
  for k,n being Nat holds
```

```
        (k + n = 0 implies k = 0 & n = 0) & :: NAT_1:23
        (k ≤ n + 1 implies k ≤ n or k = n + 1) & :: NAT_1:26
        (n ≤ k & k ≤ n + 1 implies n = k or k = n + 1) & :: NAT_1:27
        (k ≤ n implies ex m being Nat st n = k + m) & :: NAT_1:28
        k ≤ k + n & :: NAT_1:29
        (k < n + 1 iff k ≤ n) & :: NAT_1:38
        (k · n = 1 implies k = 1 & n = 1);

theorem :: BITS1:9 :: natjunk3
 for k,n,m being Nat holds
        (k ≤ n implies k · m ≤ n · m & k · m ≤ m · n &
                       m · k ≤ n · m & m · k ≤ m · n) &
                                                        :: NAT_1:20
        (k <> 0 & ( n · k = m · k or n · k = k · m or
                       k · n = k · m ) implies n = m) & :: NAT_1:24
        (k < n implies k + m < n + m & k + m < m + n & m + k < m + n &
                                          m + k < n + m) & ::
NAT_1:36
        (k ≤ n implies k ≤ n + m) & :: NAT_1:37
        (k ≤ n & n < m or k < n & n ≤ m or k < n & n < m
                                          implies k < m);

theorem :: BITS1:10 :: Natjunk1
 for n being Nat holds
        (n <> 0 iff n > 0) &
        n ≥ 0 &
        n + 1 <> 0 &
        n + 1 > 0 &
        n + 1 ≥ 1 &
        (n > 0 iff n ≥ 1) &
        (n = 0 or n = 1 or n > 1) &
        (n > 0 iff (n - 1) is Nat) &
        (n > 0 iff (n - 1) ≥ 0) &
        (n > 1 iff (n - 1) ≥ 1);

theorem :: BITS1:11 :: Natjunk2
 for n1, n2 being Nat holds
        n1 + n2 = n2 + n1 &
        (n1 + n2 = 0 iff n1 = 0 & n2 = 0) &
        (n1 < n2 + 1 iff n1 ≤ n2) &
        (n1 + 1 ≤ n2 iff n1 < n2) &
        (n1 - 1 < n2 iff n1 ≤ n2) &
        (n1 ≤ n2 - 1 iff n1 < n2) &
        n1 + n2 ≥ n1 &
        n1 + n2 ≥ n2 &
        (n1 > n2 implies n1 > 0 & n1 <> 0);
```

```
theorem :: BITS1:12 :: Catuniq
 for a, b, c, d being FinSequence st
        a^b = c^d & len a = len c & len b = len d holds a = c & b = d;

definition
        let F be non empty set;
        let A be non empty Subset of F;
        redefine mode Element of A -> Element of F;
end;

definition
        let D be non empty set;
        let F be FinSequence-DOMAIN of D;
        let A be non empty Subset of F;
        redefine mode Element of A -> Element of F;
end;

definition
        let D be non empty set;
        let X1,X2 be FinSequence-DOMAIN of D;
        let x be Element of [:X1,X2:];
        redefine func x`1 -> Element of X1;
        func x`2 -> Element of X2;
end;

definition
        let D be non empty set;
        let X1 be FinSequence-DOMAIN of D;
        let Y be non empty Subset of X1;
        let X2 be FinSequence-DOMAIN of D;
        redefine func [:Y,X2:] -> non empty Subset of [:X1,X2:];
end;

definition
        let D be non empty set;
        let X1,X2 be FinSequence-DOMAIN of D;
        let Y1 be non empty Subset of X1;
        let Y2 be non empty Subset of X2;
        redefine func [:Y1,Y2:] -> non empty Subset of [:X1,X2:];
end;

definition
        let D1, D2 be non empty set;
        let X be non empty Subset of [: D1, D2 :];
        redefine mode Element of X -> Element of [: D1, D2 :];
```

```
end;

definition
        let X be set;
        cluster -> Function-like Element of X*;
        :: for x being Element of X* holds x is Function-like
end;

definition
        let X be set;
        cluster -> FinSequence-like Element of X*;
end;

definition
        let D be non-empty set;
        let p, q be Element of D*;
        redefine func p^q -> Element of D*;
end;


theorem :: BITS1:13 :: FinSjunk
 for p being FinSequence for k being Nat holds
        (k ∈ dom p iff 1 ≤ k & k ≤ len p) &
        (1 ≤ k & k < len p implies k ∈ dom p);

definition
        let D be non empty set of Nat;
        redefine mode Element of D -> Nat;
end;

definition
        cluster non empty FinSequence;
end;

definition
        let p be FinSequence;
        redefine func dom p -> set of Nat;
end;

definition
        let D be DOMAIN;
        let p be FinSequence of D;
        redefine func dom p -> set of Nat;
end;

definition
```

```
        let p be non empty FinSequence;
        redefine func dom p -> non empty set of Nat;
end;

theorem :: BITS1:14 :: NEFjunk
 for p being FinSequence holds
        (len p <> 0 iff p is non empty) &
        (len p > 0 iff p is non empty) &
        (p is non empty implies
        1 ∈ dom p & 1 is Element of dom p
        & (len p) ∈ dom p & (len p) is Element of dom p);

theorem :: BITS1:15 :: NEFjunk2
 for p being non empty FinSequence for
        q being FinSequence holds
        (p ^ q).1 = p.1;

definition
        let D be non empty set;
        cluster non empty FinSequence of D;
end;

definition
        let D be DOMAIN;
        let p be non empty FinSequence of D;
        redefine func dom p -> non empty set of Nat;
end;

definition
        let D be DOMAIN;
        let p be non empty FinSequence of D;
        let k be Element of dom p;
        redefine func p.k -> Element of D;
end;

definition
        let p be non empty FinSequence;
        let q be FinSequence;
        redefine func p^q -> non empty FinSequence;
end;

definition
        let p be FinSequence;
        let q be non empty FinSequence;
        redefine func p^q -> non empty FinSequence;
end;
```

```
definition
        let D be DOMAIN;
        let p be non empty FinSequence of D;
        let q be FinSequence of D;
        redefine func p^q -> non empty FinSequence of D;
end;

definition
        let D be DOMAIN;
        let p be FinSequence of D;
        let q be non empty FinSequence of D;
        redefine func p^q -> non empty FinSequence of D;
end;

theorem :: BITS1:16 :: NEFoDjunk
 for D being DOMAIN for p being FinSequence of D
        for e being Element of D holds
        (len p > 0 iff p is non empty) &
        (p is non empty implies p.1 is Element of D) &
        (p ^ <* e *>) is non empty FinSequence of D &
        (<* e *> ^ p) is non empty FinSequence of D &
        (p is non empty implies p.1 is Element of D);

theorem :: BITS1:17 :: NEFoDs
 for D being DOMAIN for p being non empty FinSequence of D holds
        p.1 is Element of D &
        ex y being Element of D st y = p.1;

scheme FinRecExd{D() -> DOMAIN,
                 A() -> (Element of D()),
                 N() -> Nat,
                 P[Any,Any,Any]}:
        ex p being FinSequence of D() st N() = 0 or p.1 = A() &
             for n being Nat st 1 ≤ n & n ≤ N()-1 holds P[n,p.n,p.(n+1)]
provided
      for n being Nat st 1 ≤ n & n ≤ N()-1 holds
                        for x being Element of D()
                                ex y being Element of D() st P[n,x,y] and
      for n being Nat st 1 ≤ n & n ≤ N()-1 holds
                        for x,y1,y2 being Element of D() st
                                        P[n,x,y1] & P[n,x,y2] holds y1 = y2
;

scheme FinRecExd1{D() -> DOMAIN,
                  A() -> (Element of D()),
```

```
                N() -> Nat,
                P[Any,Any,Any]}:
        N() = 0 or
        ex p being non empty FinSequence of D() st len p = N() & p.1 = A()
&
                for n being Element of dom p st n < N() holds
P[n,p.n,p.(n+1)]
provided
      for n being Nat st 1 ≤ n & n < N() holds for x being Element of D()
                                ex y being Element of D() st P[n,x,y] and
      for n being Nat st 1 ≤ n & n < N() holds
                        for x,y1,y2 being Element of D() st
                                        P[n,x,y1] & P[n,x,y2] holds y1 = y2
;

scheme FinRecUnd1{ D()->DOMAIN,
                A() -> (Element of D()),
                N() -> Nat,
                F() -> (non empty FinSequence of D()),
                G() -> (non empty FinSequence of D()),
                P[Any,Any,Any]}:
        F() = G()
provided
      for n being Nat st 1 ≤ n & n < N()
                for x,y1,y2 being Element of D() st
                                P[n,x,y1] & P[n,x,y2] holds y1 = y2 and
      len F() = N() & F().1 = A() &
                        for n being Element of dom F() st n < N() holds
                                                P[n,F().n,F().(n+1)] and
      len G() = N() & G().1 = A() &
                        for n being Element of dom G() st n < N() holds
                                                P[n,G().n,G().(n+1)];

theorem :: BITS1:18 :: n1subn2
 for n1, n2 being Nat st n1 ≥ n2 holds n1 - n2 is Nat;

theorem :: BITS1:19 :: Catuniq1
 for a, b, c, d being FinSequence
        st len a = len c & a <> c holds a^b <> c^d;

theorem :: BITS1:20 :: Catuniq2
 for a, b, c, d being FinSequence
        st len a = len c & b <> d holds a^b <> c^d;

theorem :: BITS1:21 :: B_diff_len
 for a,b being FinSequence st len a <> len b holds a <> b;
```

```
:: set BITS = {0,1}*;
```

A.2 Bit strings model

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: Constants and functions in Mizar and their properties
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

definition
        mode Bits is Element of {0,1}*;
end;

definition
        cluster non empty Bits;
end;

definition :: Bits_0_def
        func '0 -> non empty Bits means
:: BITS1: def 1
 it = <*0*>;
end;

theorem :: BITS1:22 :: C_zero
 '0 is Bits;

definition
        let b be non empty Bits;
        let k be Element of dom b;
        redefine func b.k -> Nat;
end;

definition :: Bits_1_def
        func '1 -> non empty Bits means
:: BITS1: def 2
 it = <*1*>;
end;

theorem :: BITS1:23 :: C_one
 '1 is Bits;

definition :: nil_def
        func nil -> Bits means
:: BITS1: def 3
 it = <∅>;
```

```
end;

theorem :: BITS1:24 :: C_nil
 nil is Bits;

definition :: cat_def
        let p, q be Bits;
        redefine func p ^ q -> Bits;
end;

definition :: cat_def
        let p be non empty Bits;
        let q be Bits;
        redefine func p ^ q -> non empty Bits;
end;

definition :: cat_def
        let p be Bits;
        let q be non empty Bits;
        redefine func p ^ q -> non empty Bits;
end;

theorem :: BITS1:25 :: Bits_nil
 nil = <φ> &
        len nil = 0 &
        dom nil = φ &
        (for x being Bits holds len nil ≤ len x) &
        (for x being Bits holds nil ^ x = x & x ^ nil = x);

definition :: split_def
        let p, q be Bits;
        func p ^' q -> Element of [: {0,1}*, {0,1}* :] means
:: BITS1: def 4
 it'1 ^ it'2 = p &
        (len q ≤ len p implies len it'1=len q & len it'2=len p-len q) &
                (len q ≥ len p implies len it'1 = len p & it'2 = nil);
end;

::::::::::::::::::::::::::::::::::::::::::::::::::::
:: Primitive programs in Mizar-C
::::::::::::::::::::::::::::::::::::::::::::::::::::

theorem :: BITS1:26 :: C_nil_or_not
 for b being Bits holds b = nil or b <> nil;

theorem :: BITS1:27 :: C_unit_0_or_1
```

```
 for b being Bits st len b = 1 holds
        b = '0 or b = '1;

theorem :: BITS1:28 :: C_cat
 for p, q being Bits ex r being Bits st r = p ^ q;

:: This is commented out as there is no Cartesian product in Mizar-C
:: theorem c_split: for p,q being Bits ex r being Element
of[:{0,1}*,{0,1}*:]
::                 st p ^' q = r;

theorem :: BITS1:29 :: C_split
 for p, q being Bits ex r1, r2 being Bits
        st p ^' q = [r1, r2];

scheme C_Comp_Ind { D() -> non empty set,
                    F(Element of D()) -> Nat,
                    P[Element of D()] }:
        for a being Element of D() holds P[a]
provided
 for a being Element of D() st for a' being Element of D() st F(a') < F(a)
        holds P[a'] holds P[a];

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: New Facts about Bits
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

theorem :: BITS1:30 :: Bits_is_nil
 for x being Bits holds
        (x = nil iff x = <φ>) &
        (x = nil iff len x = 0) &
        (x = nil iff dom x = φ) &
        (x = nil iff for y being Bits holds len x ≤ len y) &
        (x = nil iff for y being Bits holds x ^ y = y) &
        (x = nil iff for y being Bits holds y ^ x = y) &
        (x = nil iff
        (ex y being Bits st
        x ^ y = y or y ^ x = y or
        (y <> nil & y ^' x = [nil,y])));

theorem :: BITS1:31 :: Bits_is_not_nil
 for a being Bits holds
        (a <> nil iff len a <> 0) &
        (a <> nil iff len a > 0) &
        (a <> nil iff len a ≥ 1) &
        (a <> nil iff (len a = 1 iff len a < 2)) &
```

```
        (a <> nil iff 1 ∈ dom a);

theorem :: BITS1:32 :: Bits_cat
 for a, b, c being Bits st a = b ^ c holds
        (len a = len b + len c) &
        (len b = 0 iff len a = len c) &
        (b = nil iff a = c) &
        (len c = 0 iff len a = len b) &
        (c = nil iff a = b) &
        (len a = 0 iff len b = 0 & len c = 0) &
        (a = nil iff b = nil & c = nil) &
        dom b c= dom a &
        (for k being Nat st k ∈ dom b holds k ∈ dom a) &
        (for k being Nat st k ∈ dom c holds (len b + k) ∈ dom a) &
        (for k being Nat st k ∈ dom b holds a.k = b.k) &
        (for k being Nat st k ∈ dom c holds a.(len b + k) = c.k) &
        len a ≥ len b &
        len a ≥ len c;

theorem :: BITS1:33 :: Bits_split
 for a, b, c, d being Bits st a ^' b = [c, d] holds
        a = c ^ d &
:: stuff from Bits_cat
        a = c ^ d &
        len a = len c + len d &
        len a = len d + len c &
        (len c = 0 iff len a = len d) &
        (c = nil iff a = d) &
        (len d = 0 iff len a = len c) &
        (d = nil iff a = c) &
        (len a = 0 iff len c = 0 & len d = 0) &
        (a = nil iff c = nil & d = nil) &
        dom c c= dom a &
        (for k being Nat st k ∈ dom c holds k ∈ dom a) &
        (for k being Nat st k ∈ dom d holds (len c + k) ∈ dom a) &
        (for k being Nat st k ∈ dom c holds a.k = c.k) &
        (for k being Nat st k ∈ dom d holds a.(len c + k) = d.k) &

        len d ≤ len a &
        len c ≤ len a &
        (len b ≤ len a iff len c = len b) &
        (len b < len a iff len d > 0) &
        (len b < len a iff len c = len b & len d > 0) &
        len c ≤ len b &
        (len c = len a or len c = len b) &
        (0 < len b & 0 < len a iff len d < len a) &
```

```
        (len b ≥ len a iff len d = 0) &
        (len b ≥ len a iff d = nil) &
        (len b ≥ len a iff c = a) &
        (len b ≥ len a iff c = a & d = nil) &
        (len b = 0 implies d = a & len c = 0) &
        (len c = 0 & len d > 0 implies len b = 0);

theorem :: BITS1:34 :: Bits_0
 '0 = <*0*> &
        len '0 = 1 &
        len '0 <> 0 &
        len '0 <> len nil &
        len '0 > 0 &
        len '0 > len nil &
        len '0 ≥ 1 &
        1 ∈ dom '0 &
        '0.1 = 0 &
        '0 is non empty Bits &
        '0 = nil ^ '0 &
        '0 = '0 ^ nil &
        '0 ^' nil = [nil,'0] &
        '0 ^' '0 = ['0,nil];

definition
        redefine func '0 -> non empty Bits;
end;

theorem :: BITS1:35 :: Bits_1
 '1 = <*1*> &
        len '1 = 1 &
        len '1 <> 0 &
        len '1 <> len nil &
        len '1 > 0 &
        len '1 > len nil &
        len '1 ≥ 1 &
        1 ∈ dom '1 &
        '1.1 = 1 &
        '1 is non empty Bits &
        '1 = nil ^ '1 &
        '1 = '1 ^ nil &
        '1 ^' nil = [nil,'1] &
        '1 ^' '0 = ['1,nil];

definition
        redefine func '1 -> non empty Bits;
end;
```

```
theorem :: BITS1:36 :: Bits_junk1
 for a being Bits holds
        a ^ nil = a &
        nil ^ a  = a &
        a ^' nil = [nil,a] &
        nil ^' a = [nil,nil] &
        len (a ^ nil) = len a &
        len (nil ^ a) = len a &
        len (a ^ '0) = len a + 1 &
        len (a ^ '0) > 0 &
        len ('0 ^ a) = len a + 1 &
        len ('0 ^ a) > 0 &
        len (a ^ '1) = len a + 1  &
        len (a ^ '1) > 0 &
        len ('1 ^ a) = len a + 1 &
        len ('1 ^ a) > 0 &
        (a ^ '0) is non empty Bits &
        (a ^ '0) is non empty Bits &
        ('0 ^ a) is non empty Bits &
        ('0 ^ a) is non empty Bits &
        (a ^ '1) is non empty Bits &
        (a ^ '1) is non empty Bits &
        ('1 ^ a) is non empty Bits &
        ('1 ^ a) is non empty Bits &
        (a is non empty iff len a > 0) &
        (a is non empty iff a <> nil) &
        ('0 ^ a).1 = 0 &
        ('1 ^ a).1 = 1 &
        (for k being Nat st k ∈ dom a holds
            ('0 ^ a).(k+1) = a.k & ('1 ^ a).(k+1) = a.k);

theorem :: BITS1:37 :: Bits_junk2
 for a,b being Bits holds
        (len b ≥ len a implies a ^' b = [a,nil]) &
        (a <> nil implies (len b ≥ len a iff a ^' b = [a,nil])) &
        len a ≤ len (a ^ b) &
        len a ≤ len (b ^ a);

theorem :: BITS1:38 :: Bits_split_0
 for a, c, d being Bits
        st a <> nil & a ^' '0 = [c, d] holds
        len c = 1 &
        len c ≥ 1 &
        len c <> 0 &
        c <> nil &
```

```
        1 ∈ dom c &
        c.1 = a.1 &
        len d < len a &
        len d + 1 = len a;

theorem :: BITS1:39 :: Bits_split_nil
 for a being Bits holds
        (a^'nil)'1 = nil &
        (a^'nil)'2 = a &
        a ^' nil = [nil,a] &
        (for af,ar being Bits st a ^' nil = [af,ar] holds af = nil & ar =
a)
;
```

A.3 Constructive inference rules

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: Basic inference rules with computational contents
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

scheme C_O_I_l { P[], Q[] }:
       P[] or Q[]
provided
      P[];

scheme C_O_I_r { P[], Q[] }:
       P[] or Q[]
provided
      Q[];

scheme C_O_E { P[], Q[], R[] }:
       R[]
provided
      P[] or Q[] and
      P[] implies R[] and
      Q[] implies R[];

scheme C_I_E { P[], Q[] }:
       Q[]
provided
      P[] implies Q[] and
      P[];

scheme C_U_E { D() -> non empty set,
               B() -> Element of D(),
```

```
                P[Element of D()] }:
        P[B()]
provided
      for b being Element of D() holds P[b] and
      B() is Bits;

scheme C_E_I { S() -> non empty set,
        A() -> Element of S(),
        P[Element of S()] }:
        ex b being Element of S() st P[b]
provided
      P[A()] and
      A() is Bits;

scheme C_E_E { S() -> non empty set,
                P[Element of S()] }:
        ex a being Element of S() st P[a]
provided
      ex a being Element of S() st P[a];

scheme C_EQ { D() -> non empty set,
              A() -> Element of D(),
              D'() -> non empty Subset of D(),
              A'() -> Element of D'(),
              P[Element of D()] }:
        P[A'()]
provided
      A'() = A() and
      P[A()];

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: Derived inference rules with computational contents
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

scheme C_03_I_l { P[], Q[], R[] }:
        P[] or Q[] or R[]
provided
      P[];

scheme C_03_I_m { P[], Q[], R[] }:
        P[] or Q[] or R[]
provided
      Q[];

scheme C_03_I_r { P[], Q[], R[] }:
        P[] or Q[] or R[]
```

```
provided
      R[];

scheme C_03_E { P[], Q[], R[], S[] }:
        S[]
provided
      P[] or Q[] or R[] and
      P[] implies S[] and
      Q[] implies S[] and
      R[] implies S[];

scheme C_U2_E { D() -> non empty set,
                A() -> Element of D(),
                B() -> Element of D(),
                P[Element of D(), Element of D()] }:
        P[A(), B()]
provided
      for a,b being Element of D() holds P[a,b] and
      A() is Bits and
      B() is Bits;

scheme C_UI_E { D() -> non empty set,
                A() -> Element of D(),
                P[Element of D()], Q[Element of D()] }:
        Q[A()]
provided
      for b being Element of D() holds P[b] implies Q[b] and
      A() is Bits and
      P[A()];

scheme C_U2I_E { D() -> non empty set,
                 A() -> Element of D(),
                 B() -> Element of D(),
                 P[Element of D(), Element of D()],
                 Q[Element of D(), Element of D()] }:
        Q[A(), B()]
provided
      for a,b being Element of D() holds P[a, b] implies Q[a, b] and
      A() is Bits and
      B() is Bits and
      P[A(), B()];

scheme C_UIU_E { D() -> non empty set,
                 A() -> Element of D(),
                 B() -> Element of D(),
                 P[Element of D()],
```

```
                Q[Element of D(),Element of D()] }:
        Q[A(),B()]
provided
      for a being Element of D() st P[a] holds
            for b being Element of D() holds Q[a,b] and
      A() is Bits and
      P[A()] and
      B() is Bits;

scheme C_E2_I { D() -> non empty set,
                D'() -> non empty set,
                A() -> Element of D(),
                B() -> Element of D'(),
                P[Element of D(),Element of D'()] }:
       ex a being Element of D() st ex b being Element of D'() st P[a,b]
provided
      P[A(),B()] and
      A() is Bits and
      B() is Bits;
```

Appendix B

Bits, UNat and BNat

BITS2.ABS

January 28, 1996

```
environ
        vocabulary
        FUNC_REL, SUB_OP, REAL_1, NAT_1, INT_1, ANAL,
        FINSEQ, COORD, FUNC, FINITER2, TUPLES, PI, SIGMA, NEWTON,
        POWER, POWER1, NISHIYA, BITS;
        constructors
        BOOLE, FUNCT_2, SEQ_1;
        signature
        TARSKI, REAL_1, SUBSET_1, NAT_1, INT_1, INT_2, FUNCT_1,
        FINSEQ_1, MCART_1, DOMAIN_1, FINSEQ_2, FINSOP_1, PREPOWER, POWER,
        BINARITH, BITS1;
        theorems
        TARSKI, AXIOMS, BOOLE, REAL_1, REAL_2, SUBSET_1, NAT_1, FUNCT_1,
        FINSEQ_1, FINSEQ_2, INT_1, INT_2, MCART_1, NEWTON, PREPOWER, POWER,
        SQUARE_1, BINARITH, BITS1;
        schemes
        BOOLE, NAT_1, FINSEQ_1, RECDEF_1, BITS1;
        definitions
        TARSKI;
        clusters
        FUNCT_1, FINSEQ_1,FINSEQ_2, BITS1;
begin
```

B.1 Bits programs

```
theorem :: BITS2:1 :: C_len_decide
 for p being Bits holds for q being Bits holds
        len p < len q or len p = len q or len p > len q;
```

```
theorem :: BITS2:2 :: C_len_decide2
 for p, q being Bits
        holds len p < len q or len p ≥ len q;

theorem :: BITS2:3 :: C_eqf_or_not
 for a, b being Bits st a <> nil & b <> nil
        holds a.1 = b.1 or a.1 <> b.1;

theorem :: BITS2:4 :: C_unit_eq_or_not
 for a, b being Bits st len a = 1 & len b = 1
        holds a = b or a <> b;

theorem :: BITS2:5 :: C_eq_or_not
 for p, q being Bits holds p = q or p <> q;
```

B.2 Unary naturals

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: Unary natural definition
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

definition :: UNAT_def
        func UNAT -> non empty Subset of {0,1}* means
:: BITS2: def 1
 for x being Any holds x ∈ it iff
        ex b being Bits st x = b &
                for k being Nat st k ∈ dom b holds b.k = 0;
end;

definition
        mode UNat is Element of UNAT;
end;

theorem :: BITS2:6 :: UNat_0
 for b being Bits holds
                b is UNat iff for k being Nat st k ∈ dom b holds b.k = 0;

theorem :: BITS2:7 :: UNat_split
 for p being UNat
        for q, r1, r2 being Bits st p ^' q = [r1, r2] holds
                                                r1 is UNat & r2 is UNat;

definition :: U0_Def, U1_Def
        func U0 -> UNat means
:: BITS2: def 2
```

```
 it = nil;
        func U1 -> UNat means
:: BITS2: def 3
 it = '0;
end;

theorem :: BITS2:8 :: C_U0
 U0 is Bits;

theorem :: BITS2:9 :: C_U1
 U1 is Bits;

theorem :: BITS2:10 :: Bits_U0
 U0 = nil &
        U0 = <φ> &
        len U0 = 0 &
        dom U0 = φ &
        for x being Bits holds len U0 ≤ len x;

theorem :: BITS2:11 :: Bits_U1
 U1 = '0 &
        len U1 = 1 &
        len U1 <> 0 &
        len U1 > 0 &
        len U1 ≥ 1 &
        1 ∈ dom U1 &
        U1.1 = 0;

theorem :: BITS2:12 :: UNat_junk
 nil is UNat &
        '0 is UNat &
        (for u being UNat st len u = 1 holds u = U1);

theorem :: BITS2:13 :: UNat_len_bij
 for u, v being UNat holds u = v iff len u = len v;

definition :: U2N_def
        let u be UNat;
        func U2N u -> Nat means
:: BITS2: def 4
 it = len u;
end;

definition :: N2U_def
        let n be Nat;
        func N2U n -> UNat means
```

```
:: BITS2: def 5
 len it = n;
end;

theorem :: BITS2:14 :: U2N2U
 for u being UNat holds u = N2U U2N u;

theorem :: BITS2:15 :: N2U2N
 for n being Nat holds n = U2N N2U n;

theorem :: BITS2:16 :: UTwos
 for u being UNat for n being Nat holds
                                        U2N u = n iff u = N2U n;

theorem :: BITS2:17 :: two_junk
 U2N U0 = 0 &
        N2U 0 = U0 &
        N2U 0 = nil &
        U2N U1 = 1 &
        N2U 1 = U1 &
        N2U 1 = '0;

definition
        let p, q be UNat;
        redefine func p ^ q -> UNat;
end;

theorem :: BITS2:18 :: C_Ucat
 for a,b being UNat ex c being UNat st c = a ^ b;

definition :: Uplus_Def
        let p, q be UNat;
        func p + q -> UNat means
:: BITS2: def 6
 it = p ^ q;
end;

theorem :: BITS2:19 :: C_Uplus
 for a,b being UNat holds ex c being UNat st c = a + b;

theorem :: BITS2:20 :: Uplus
 for p, q being UNat holds
        U2N (p + q) = U2N p + U2N q;

definition
        let p be UNat;
```

```
        let q be Bits;
        redefine func p ^‘ q -> Element of [: UNAT, UNAT :];
end;

definition :: UNat_mult_def
        let u, v be UNat;
        func u · v -> UNat means
:: BITS2: def 7
 U2N u · U2N v = U2N it;
end;

theorem :: BITS2:21 :: UNat_mult_by_U0
 for v being UNat holds U0 = U0 · v & U0 = v · U0;

theorem :: BITS2:22 :: C_UNat_mult
 for u being UNat holds
        for v being UNat ex uv being UNat st uv = u · v;

theorem :: BITS2:23 :: C_Bits2UNat
 for a being Bits holds ex u being UNat st len u = len a;
```

B.3 Binary naturals

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: Binary naturals
::::::::::::::::::::::::::::::::::::::::::::::::::::::

theorem :: BITS2:24 :: B2N_exist
 for b being Bits holds len b = 0 or
            ex p being non empty FinSequence of NAT st len p = len b &
                p.1 = b.1 & (for n being Element of dom p st n < (len b)
holds
                  ex bn being Nat st bn = b.(n+1) & p.(n+1) = 2 · p.n + bn)
;

definition :: B2N_Def
        let b be Bits;
        func B2N b -> Nat means
:: BITS2: def 8

        (len b = 0 implies it = 0) &
        (len b > 0 implies
            ex p being non empty FinSequence of NAT st
                it = p.(len b) & len p = len b & p.1 = b.1 &
                for n being Element of dom p st n < len b holds
```

```
                    ex bn being Nat st
                        bn = b.(n+1) & p.(n+1) = 2 · p.n + bn);
end;

theorem :: BITS2:25 :: B2N_junk0

        B2N nil = 0 &
        B2N '0 = 0 &
        B2N '1 = 1;

theorem :: BITS2:26 :: B2N_def
 B2N nil = 0 &
        (for b being Bits holds
                B2N (b ^ '0) = 2 · B2N b &
                B2N (b ^ '1) = 2 · B2N b + 1);

definition :: {
        let a,n be Nat;
        redefine func a #N n -> Nat;
end;

theorem :: BITS2:27 :: powerjunk1
 for a being Real holds
        a #N 0 = 1 &
        a #N 1 = a;

theorem :: BITS2:28 :: powerjunk2
 for a being Real for n being Nat holds
        (a > 1 implies a #N n < a #N (n+1));

theorem :: BITS2:29 :: power2
 for k,n being Nat holds k < n iff 2 #N k < 2 #N n;

theorem :: BITS2:30 :: power2'
 for k,n being Nat holds k ≤ n iff 2 #N k ≤ 2 #N n;

theorem :: BITS2:31 :: power21
 for k being Nat holds
        2 #N k ≥ 1 &
        2 #N k > 0 &
        (k > 0 implies 2 #N k ≥ 2);

theorem :: BITS2:32 :: B2N_def2
 for b being Bits holds
        B2N ( nil ) = 0 &
        B2N ('0 ^ b) = B2N b &
```

```
        B2N ('1 ^ b) = 2 #N len b + B2N b;

theorem :: BITS2:33 :: UNat_B2N
 for u being UNat holds
        B2N u = 0 & for a being Bits holds B2N(u ^ a) = B2N a;

theorem :: BITS2:34 :: C_Pad_zero
 for a,b being Bits st len a ≥ len b holds
          ex c being Bits st len c = len a & B2N c = B2N b;

theorem :: BITS2:35 :: Bits_length
 for a being Bits holds B2N a < 2 #N len a;

definition :: BNAT_def
        func BNAT -> non empty Subset of {0,1}* means
:: BITS2: def 9
 for x being Any holds x ∈ it iff
        ex b being Bits st x = b & (len b = 0 or b.1 = 1);
end;

definition :: {
        mode BNat is Element of BNAT;
end;

definition
        cluster non empty BNat;
end;

theorem :: BITS2:36 :: BNat_def
 for b being Bits holds b is BNat iff (len b = 0 or b.1 = 1);

theorem :: BITS2:37 :: BNat_split_0
 for a being BNat st a <> nil for b,c being Bits st
        a ^' '0 = [b,c] holds b = '1;

definition :: B0_Def, B1_Def
        func B0 -> BNat means
:: BITS2: def 10
 it = nil;
        func B1 -> non empty BNat means
:: BITS2: def 11
 it = '1;
end;

theorem :: BITS2:38 :: C_B0
 B0 is Bits;
```

```
theorem :: BITS2:39 :: C_B1
 B1 is Bits;

theorem :: BITS2:40 :: BNat_length
 for a being BNat st a <> B0 holds
                                        B2N a ≥ 2 #N (len a -' 1);

theorem :: BITS2:41 :: Bits_B0
 B0 = nil &
        B2N B0 = 0 &
        for b being BNat st B2N b = 0 holds b = B0;

theorem :: BITS2:42 :: Bits_B1
 B1 = '1 &
        B1 = nil ^ '1 &
        B1 = B0 ^ '1 &
        B2N B1 = 1 &
        for b being BNat st B2N b = 1 holds b = B1;

definition :: {
        cluster non empty BNat;
end;

definition :: {
        let p be non empty BNat;
        let q be Bits;
        assume  p <> B0;
        redefine func p ^ q -> non empty BNat;
end;

theorem :: BITS2:43 :: B2N_unique
 for k being Nat holds
        for b1,b2 being Bits st len b1 = k & len b2 = k &
            B2N b1 = B2N b2 holds b1 = b2;

theorem :: BITS2:44 :: BNat_len
 for a being BNat for b being Bits st len a > len b holds
        B2N a > B2N b;

definition :: N2B_Def
        let n be Nat;
        func N2B n -> BNat means
:: BITS2: def 12
 B2N it = n;
end;
```

```
theorem :: BITS2:45 :: NAT_BNAT_NAT
 for n being Nat holds B2N N2B n = n;

theorem :: BITS2:46 :: BNAT_NAT_BNAT
 for b being BNat holds N2B B2N b = b;

theorem :: BITS2:47 :: Bits2BNat
 for a being Bits ex b being BNat st B2N a = B2N b;

theorem :: BITS2:48 :: B2N_cat
 for a,b being Bits holds
       B2N(a ^ b) =  B2N a · 2 #N len b + B2N b;

theorem :: BITS2:49 :: B2N_junk2
 for a,b being Bits holds
       B2N(b ^ a) ≥ B2N a;
```

Appendix C

Arithmetic operations on BNat

BITS3.ABS

January 28, 1996

```
environ
        vocabulary
        FUNC_REL, SUB_OP, REAL_1, NAT_1, INT_1, ANAL,
        FINSEQ, COORD, FUNC, FINITER2, TUPLES, PI, SIGMA, NEWTON,
        POWER, POWER1, NISHIYA, BITS;
        constructors
        BOOLE, FUNCT_2, SEQ_1;
        signature
        TARSKI, REAL_1, SUBSET_1, NAT_1, INT_1, INT_2, FUNCT_1,
        FINSEQ_1, MCART_1, DOMAIN_1, FINSEQ_2, FINSOP_1, PREPOWER, POWER,
        BINARITH, BITS1, BITS2;
        theorems
        TARSKI, AXIOMS, BOOLE, REAL_1, REAL_2, SUBSET_1, NAT_1, FUNCT_1,
        FINSEQ_1, FINSEQ_2, INT_1, INT_2, MCART_1, NEWTON, PREPOWER, POWER,
        SQUARE_1, BINARITH, BITS1, BITS2;
        schemes
        BOOLE, NAT_1, FINSEQ_1, RECDEF_1, BITS1;
        definitions
        TARSKI;
        clusters
        FUNCT_1, FINSEQ_1,FINSEQ_2, BITS1, BITS2;
begin

definition
        func BIT -> non empty Subset of {0,1}* means
:: BITS3: def 1
 it = {<*0*>,<*1*>};
end;

definition
```

```
        mode Bit is Element of BIT;
end;

definition
        cluster non empty Bit;
end;

definition
        redefine func '0 -> non empty Bit;
end;

theorem :: BITS3:1 :: C_0
 '0 is Bits;

definition
        redefine func '1 -> non empty Bit;
end;

theorem :: BITS3:2 :: C_I
 '1 is Bits;

theorem :: BITS3:3 :: Bit_junk0
 for b being Bits holds
            (b is Bit iff b = '0 or b = '1) &
            (b is Bit iff len b = 1);

theorem :: BITS3:4 :: Bit_junk1
 for b being Bit holds
        (b = '0 or b = '1) &
        len b = 1 &
        len b <> 0 &
        len b > 0 &
        b <> nil;

theorem :: BITS3:5 :: C_Bit_decide
 for b being Bit holds
        b = '0 or b = '1;

theorem :: BITS3:6 :: N2B_junk0
 N2B 0 = B0 &
        N2B 0 = nil &
        N2B 1 = B1;

definition :: B2B_Def
        let b be Bits;
        func B2B b -> BNat means
```

```
:: BITS3: def 2
 it = N2B B2N b;
end;

theorem :: BITS3:7 :: B2B_def
 B2B B0 = B0 &
        B2B nil = nil &
        B2B B1 = B1 &
        (for a being Bits holds
             B2B a = N2B B2N a &
             B2N a = B2N B2B a &
             len a ≥ len B2B a) &
        (for u being UNat holds B2B u = B0 & B2B u = nil) &
        (for b being BNat holds B2B b = b);
```

C.1 Binary addition

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: Binary plus
::::::::::::::::::::::::::::::::::::::::::::::::::::::

definition
        let p, q be Bits;
        func p + q -> BNat means
:: BITS3: def 3
 B2N it = B2N p + B2N q;
        commutativity;
end;

theorem :: BITS3:8 :: BNat_plus_def
 for a,b being Bits holds
        B2N(a + b) = B2N a + B2N b &
        B2N(a + b) = B2N b + B2N a &
        a + b = N2B(B2N a + B2N b) &
        a + b = N2B(B2N b + B2N a) &
        (for ab being BNat st B2N ab = B2N(a+b) holds ab = a+b);

theorem :: BITS3:9 :: BNat_plus_nil
 for b being Bits holds B2N(b+nil)=B2N b & B2N(b+B0)=B2N b &
        (b is BNat implies b + nil = b & b + B0 = b);

theorem :: BITS3:10 :: C_Bits_plus
 for a,b being Bits st len a = len b holds
            ex c being Bits st ex cu being Bit st len c = len a &
                B2N(cu ^ c) = B2N(a + b);
```

```
theorem :: BITS3:11 :: C_BNat_plus
 for p, q being Bits ex pq being BNat st pq = p + q;

theorem :: BITS3:12 :: Bplus_junk
 for a being Bits holds
        B2N(a + nil) = B2N a &
        B2N(a + '0) = B2N a &
        for u being UNat holds
            B2N(a + u) = B2N a;
```

C.2 Binary subraction

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: Binary minus
::::::::::::::::::::::::::::::::::::::::::::::::::::::

theorem :: BITS3:13 :: AsCom3
 for a,b,b2 being Real holds
        a + (b + b2) = b + (a + b2) &
        a + b + b2 = a + b2 + b;

theorem :: BITS3:14 :: AsCom4
 for a,a2,b,b2 being Real holds a + a2 + (b + b2) = a + b + (a2 + b2);

theorem :: BITS3:15 :: C_Bits_minus
 for a,b being Bits st len a = len b holds
        ex c being Bits st ex sb being Bit st len c = len a &
            B2N(sb ^ a) = B2N(b + c) &
            (B2N a < B2N b iff sb = '1);

theorem :: BITS3:16 :: B2N_notUNat
 for a being Bits st not a is UNat holds
        B2N a <> 0 &
        B2N a > 0;

definition :: {
        let p be Bits;
        func - p -> Bits means
:: BITS3: def 4
 (p is UNat implies it = nil) &
             (not p is UNat implies len it = len p &
              B2N it = 2 #N len p - B2N p);
end;
```

```
theorem :: BITS3:17 :: C_UNat_or_not
 for a being Bits holds
        a is UNat or not a is UNat;

theorem :: BITS3:18 :: C_BNatUminus
 for a being Bits holds ex b being Bits st b = -a;

theorem :: BITS3:19 :: C_B2N_junk2B
 for a,b being Bits st B2N a ≤ B2N b holds
        ex c being Bits st len c = len b & B2N c = B2N a;

theorem :: BITS3:20 :: C_B2N_decide2e
 for a,b being Bits st len a = len b holds
        B2N a < B2N b or B2N a ≥ B2N b;

theorem :: BITS3:21 :: C_B2N_decide2
 for a,b being Bits holds
        B2N a < B2N b or B2N a ≥ B2N b;

definition
        let p, q be Bits;
        func p - q -> BNat means
:: BITS3: def 5
 (B2N p ≥ B2N q implies
                B2N it = B2N p - B2N q) &
        (B2N p < B2N q implies
                B2N it = 2 #N len q + B2N p - B2N q);
end;

theorem :: BITS3:22 :: C_BNat_minus
 for p,q being Bits holds ex pq being BNat st ex s being Bit st
        pq = p - q & (B2N p < B2N q iff s = '1);
```

C.3 Binary multiplication

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: Binary times
::::::::::::::::::::::::::::::::::::::::::::::::::::::

definition
        let p, q be Bits;
        func p · q -> BNat means
:: BITS3: def 6
 it = N2B(B2N p · B2N q);
        commutativity;
```

```
end;

theorem :: BITS3:23 :: C_BNat_t
 for a,b being Bits holds
        ex c being BNat st ex u being UNat st len u = len a & c = a · b;

theorem :: BITS3:24 :: C_BNat_times
 for a,b being Bits ex c being BNat st c = a · b;

theorem :: BITS3:25 :: B1_NE
 B1 is non empty;

scheme C_U2_EX { D() -> non empty set,
                 D'() -> non empty set,
                 A() -> Element of D(),
                 B() -> Element of D'(),
                 P[Element of D(),Element of D'()] }:
        P[A(),B()]
provided
      for a being Element of D() for b being Element of D'() holds P[a,b]
and
      A() is Bits and
      B() is Bits;

scheme C_UIU_EX { D() -> non empty set,
                  D'() -> non empty set,
                  A() -> Element of D(),
                  B() -> Element of D'(),
                  P[Element of D()],
                  Q[Element of D(),Element of D'()] }:
        Q[A(),B()]
provided
      for a being Element of D() st P[a] holds
            for b being Element of D'() holds Q[a,b] and
      A() is Bits and
      P[A()] and
      B() is Bits;

theorem :: BITS3:26 :: BNat_times_def
 for a,b being Bits holds
        B2N(a·b) = B2N a · B2N b;

theorem :: BITS3:27 :: BNat_dist
 for a,b,c being Bits holds a·(b+c) = (a·b) + (a·c);
```

C.4 Binary division

```
::::::::::::::::::::::::::::::::::::::::::::
:: Binary division
::::::::::::::::::::::::::::::::::::::::::::

theorem :: BITS3:28 :: Div0
 for n being Nat holds
        n div 0 = 0 &
        n mod 0 = 0;

definition
        let a,b be Bits;
        func a / b -> Element of [: BNAT, BNAT :] means
:: BITS3: def 7
 it`1 = N2B(B2N a div B2N b) &
        it`2 = N2B(B2N a mod B2N b);
end;

theorem :: BITS3:29 :: Bdiv_def
 for a,b being Bits holds
        (a/b)`1 = N2B(B2N a div B2N b) &
        B2N(a/b)`1 = B2N a div B2N b &
        (a/b)`2 = N2B(B2N a mod B2N b) &
        B2N(a/b)`2 = B2N a mod B2N b &
        (0 < B2N b implies B2N a = B2N b · B2N(a/b)`1 + B2N(a/b)`2) &
        (B2N a < B2N b implies (a/b)`1=B0 & (a/b)`2=B2B a & a/b=[B0,B2B a])
&
        for q,r being BNat st B2N a =B2N(q·b+r) & B2N r < B2N b holds
            (a/b)`1 = q &
            (a/b)`2 = r &
            a/b = [q,r] &
            (a is BNat implies a = q · b + r) &
            (B2N a < B2N b implies q = B0 & r = B2B a);

theorem :: BITS3:30 :: Bdiv_junk
 for a,b being Bits st B2N b > 0 holds
        for q,r being BNat st a/b = [q,r] holds
            B2N r < B2N b &
            B2N a = B2N q · B2N b + B2N r &
            B2N a = B2N(q · b + r) &
            B2B a = q · b + r &
            (a is BNat implies a = q · b + r);

theorem :: BITS3:31 :: N2B_junk0
 N2B 0 = B0 &
```

```
        N2B 0 = nil;

theorem :: BITS3:32 :: C_Bcat
 for a being BNat st a is non empty for b being Bits holds
        ex ab being BNat st ab = a ^ b;

definition
        let a be BNat;
        let b be Bits;
        redefine func a ^‘ b -> Element of [:BNAT,{0,1}*:];
end;

theorem :: BITS3:33 :: C_Usplit
 for u being UNat for b being Bits holds
        ex uf,ur being UNat st u ^‘ b = [uf,ur];

theorem :: BITS3:34 :: C_Bsplit
 for a being BNat for b being Bits holds
        ex af being BNat st ex ar being Bits st a ^‘ b = [af,ar];

theorem :: BITS3:35 :: C_BNat_minus2
 for a,b being Bits holds ex ab being BNat st
        ab = a - b;

theorem :: BITS3:36 :: C_BNat_div
 for a,b being BNat holds ex q,r being BNat st
        a / b = [q,r];
```